JAVA SOUND ALS PLATTFORM FÜR DIE ENTWICKLUNG STUDIOTAUGLICHER AUDIOAPPLIKATIONEN

Diplomarbeit

an der

Fachhochschule Stuttgart –

Hochschule der Medien

Manuel René Robledo Esparza

Erstprüfer:	Prof. Dr. Fridtjof Toenniessen
Zweitprüfer:	Dipl.-Ing. (FH) Tobias Frech

Bearbeitungszeitraum: 15. April 2004 bis 14. August 2004

Eingereicht am: 09. August 2004

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort, Datum

Unterschrift

"Currently, we are enabling multimedia in the computer desktop market by adding true sound support in the Java 2 platform. In the future, we would like to see our Java Sound API technology used in professional, consumer and Internet audio applications."

Michael Bundschuh,
Java Media Manager,
1999

"1.5 ist die erste Version, die studiotaugliche Anwendungen ermöglicht."

Florian Bömers,
Java Sound Chefentwickler,
2004

Kurzfassung

Die vorliegende Arbeit versucht, die Eignung von Java Sound als Basisplattform für die Entwicklung studiotauglicher Audioanwendungen zu evaluieren. Dafür werden die Besonderheiten der Audioprogrammierung mit Sampling- und MIDI-Daten erarbeitet und daraus Anforderungen an Audiosoftware für den Studiobereich abgeleitet. Es wird kritisch überprüft, inwieweit Java und Java Sound theoretisch geeignet sind, diese Anforderungen zu erfüllen. Gegenstand der Überprüfung ist dabei sowohl das Java Sound API als auch dessen Implementierungen. Im praktischen Teil werden zwei Sampleeditoren und einige PlugIns entwickelt, um aus der praktischen Erfahrung weitere Ergebnisse zu gewinnen. Abschließend wird eine Gesamtbewertung der Studiotauglichkeit von Java Sound vorgenommen und ein Zukunftsausblick gegeben.

Schlagwörter: Java Sound, Audio, MIDI, Studio, professionell

Abstract

This work is intended to evaluate the suitability of the Java Sound framework for being used as a basis for the development of professional audio applications that can be used in a studio environment. The main principles of audio programming including sampling and MIDI are shown. The main requirements for audio software in studio environments will then be derived from those principles, followed by a theoretical verification of Java's and Java Sound's ability to fulfil those requirements. Both the Java Sound API and its implementations will be taken into account. In the practical part of this work, two sample editors and some plug-ins are developed to gather additional results from the practical experience. At the end the results are summarized in a rating about Java Sound's suitability as a development platform for professional audio applications and an outlook into future development of Java Sound is given.

Keywords: Java Sound, Audio, MIDI, Studio, professional

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele

Abkürzungsverzeichnis

AIFF	Audio Interchange File Format
ALSA	Advanced Linux Sound Architecture
AMT	Active MIDI Transmission
API	Application Programming Interface
ASIO	Audio Streaming Input Output
AWT	Abstract Window Toolkit
BPM	Beats per Minute
CPU	Central Processing Unit
DIN	Deutsches Institut für Normung
DSP	Digital Signal Processor / Digital Signal Processing
EASI	Enhanced Audio Streaming Interface
FFT	Fast Fourier Transformation
GC	Garbage Collector / Garbage Collection
GCJ	GNU Java Compiler
GM	General MIDI
GNU	GNU´s Not UNIX
GSIF	Giga Sampler Interface
GUI	Graphical User Interface
Hz	Hertz
ISA	Integrated System Architecture
J2SE	Java 2 Standard Edition
JDK	J2SE Development Kit
JDS	Java Desktop System
JIT	Just-In-Time
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LGPL	Lesser General Public License
LSB	Least Significant Byte
LTB	Linear Time Base

MIDI	Musical Instrument Digital Interface
MMA	MIDI Manufacturer's Association
MPEG	Moving Pictures Expert Group
MSB	Most Significant Byte
MTC	MIDI Time Code
OSS	Open Sound System
PCI	Peripheral Component Interconnect
PCM	Pulse Code Modulation
RFE	Request for Enhancement
RAM	Random Access Memory
RMF	Rich Media Format
ROM	Read Only Memory
SAOL	Structured Audio Orchestra Language
SASL	Structured Audio Score Language
SDK	Software Development Kit
SMPTE	Society of Motion Picture and Television Engineers
SPI	Service Provider Interface
SP-MIDI	Scalable Polyphony MIDI
SWT	Standard Widget Toolkit
WDM	Windows Driver Model
XMF	eXtensible Music Format

Vorwort

Die Motivation zu dieser Arbeit entstand, als ich für meine Groovebox (Roland MC-909) versuchte, einen Softwareeditor zu schreiben. Um plattformunabhängig zu bleiben, sollte dieser in Java realisiert werden. Ich musste schnell feststellen, dass Java Sound zwar von der API[1]-Schnittstelle her eine umfangreiche Unterstützung für den Zugriff auf Audio- und MIDI[2]-Hardware bietet, die Implementierung in J2SE[3] 1.4.0, welche ich damals benutzte, aber unvollständig ist.

Ich interessierte mich dafür, inwieweit ein Projekt wie der von mir geplante Editor bzw. allgemein professionelle Anwendungen im Audiobereich mit Java möglich sind. Hinzu kam für mich der Wunsch, die Besonderheiten im Bereich der Audioprogrammierung theoretisch und praktisch zu erlernen und zu erfahren.

Im Verlauf meines Studiums erlernte ich wichtige Grundlagen, allerdings konzentrierten sich diese meistens entweder auf Informatik- oder Medieninhalte. Da mich insbesondere die Kombination dieser beiden Bereiche interessiert, bot sich ein Diplomarbeitsthema an, das diese Kombination beinhaltet. Ich konnte im Rahmen dieser Arbeit feststellen, dass ich an dieser Thematik Freude finde und werde mich daher auch über den Rahmen der Diplomarbeit hinaus damit beschäftigen.

Herzlich bedanken möchte ich mich bei Florian Bömers für das Interview und die Unterstützung, bei Matthias Pfisterer für die kompetente Hilfe, bei meinen Korrektoren Prof. Dr. Fridtjof Toenniessen und Dipl.-Ing. (FH) Tobias Frech für die Betreuung sowie bei meiner Freundin, meinen Freunden und meiner Familie für Geduld und Hilfe.

In dieser Arbeit werden folgende Elemente durch besondere Schriftarten hervorgehoben:

- `Codeteile, Datei- und Verzeichnisnamen`
- Tasten
- Internetadressen
- Texte auf dem Bildschirm
- Buttons auf dem Bildschirm

1 Application Programming Interface
2 Musical Instrument Digital Interface
3 Java 2 Standard Edition

I. Einleitung

Längst hat Java den Ruf abgeschüttelt, eine reine Internetprogrammiersprache zu sein und vornehmlich auf die Programmierung von Applets ausgerichtet zu sein. Zwar liegt der Haupteinsatzbereich nach wie vor nicht bei Desktopanwendungen, sondern viel mehr im Serverbereich, aber die bekannten Vorteile von Java, wie beispielsweise Plattformunabhängigkeit und Sicherheit, lassen auch auf dem Desktop immer mehr Applikationen auf Java-Basis entstehen. Hinzu kommen neue Bestrebungen von Sun, wie das Java Desktop System (JDS)[4], um Java „desktopfähiger" zu machen.

Relativ wenig konnte sich Java dagegen bisher im Bereich der Verarbeitung audiovisueller Medien durchsetzen. Dies mag zum einen daran liegen, dass Java lange der Ruf anhaftete, langsam zu sein; zum anderen gab es in der Sun-Implementierung von Java Sound bis zuletzt unzählige Lücken und Bugs, so z.B. die mangelhafte Unterstützung externer Hardware. Diese erschwerten es zumindest, Audioapplikationen zu entwickeln, die den Ansprüchen professioneller Anwender aus dem Studiobereich gerecht werden können.

Zwar haben verschiedene Projekte in den letzten Jahren gezeigt, dass mit Java brauchbare Anwendungen im Audiobereich, wie MP3-Player oder einfache MIDI-Applikationen mit Hilfe von JMF[5] oder Java Sound möglich sind. Für den Studiobereich existieren jedoch bis dato nur wenige Projekte, die in Java realisiert sind. Diese umgehen zudem alle mehr oder weniger stark die von Java angebotenen Schnittstellen durch Eigenimplementierungen.

Seit Anfang 2004 ist die Beta-Version des JDK[6] 1.5.0 verfügbar. Darin scheinen viele der Mängel beseitigt und ein großer Schritt nach vorne getan zu sein. Ob allerdings das Java Sound API und dessen Implementierung nun tatsächlich die Entwicklung studiotauglicher Audioanwendungen ermöglicht, und ob die von Sun angestrebte Plattformunabhängigkeit auch bei der sonst sehr hardwarenahen Audioprogrammierung gewährleistet bleibt, muss sich noch zeigen.

Im Folgenden soll versucht werden, diese Fragen durch theoretische Untersuchung sowie durch Beispielapplikationen im praktischen Teil zu erörtern.

---

4 http://java.com/en/business/products/jds/

5 Java Media Framework

6 J2SE Development Kit

II. Ziel und Aufbau dieser Arbeit

Primäres Ziel dieser Arbeit ist es, die Java-Sound-Technologie von theoretischer und praktischer Seite her zu beleuchten. Dabei soll kritisch überprüft werden, ob sie den besonderen Anforderungen an Audiosoftware im Studioumfeld gerecht werden kann und inwieweit dabei die Plattformunabhängigkeit gewahrt bleibt.

Zunächst wird ein allgemeiner Einblick in die Thematik der Audioprogrammierung gegeben. Hierzu zählen grundlegende Ansätze und Schwierigkeiten, die diese mit sich bringt. Ebenso wird herausgestellt, welches die Anforderungen an studiotaugliche Audiosoftware sind.

Der folgende Teil befasst sich mit den Grundvoraussetzungen, welche Java für die Entwicklung von Audiosoftware bereitstellt. Anschließend wird die Java-Sound-Technologie von der Idee über die Programmierschnittstellen bis hin zum derzeitigen Stand der Implementierungen beschrieben. Hier soll bereits eine Bewertung von Java und Java Sound im Speziellen als Plattform für die professionelle Audioprogrammierung vorgenommen werden, allerdings nur anhand der bis dahin gewonnenen theoretischen Erkenntnisse.

Im praktischen Teil werden die Anwendungen, die im Rahmen dieser Diplomarbeit realisiert wurden, kurz vorgestellt und Besonderheiten herausgestellt, die sich im Bezug auf Java Sound ergeben haben. Lösungen für aufgetretene Probleme sollen ebenso aufgezeigt werden, wie die (teils betriebssystem- oder hardwareabhängigen) Ursachen für diese Probleme.

Zum Schluss werden die Ergebnisse aus theoretischer und praktischer Betrachtung der Java-Sound-Technologie zusammengefasst und kritisch auf die Tauglichkeit für den Studiobereich hin bewertet. Schließlich sollen noch die Zukunftsperspektiven von Java Sound in den Blick genommen werden.

III. Audioprogrammierung

Zunächst werden einige Grundlagen der Audioprogrammierung aufgezeigt, um darauf aufbauend das Konzept sowie die Stärken und Schwächen von Java Sound erläutern zu können.

1. Von analog zu digital

1.1. Analoge Audiotechnik

Was Menschen als Klang empfinden, ist nichts anderes als sich verändernder Luftdruck[7]. Die Moleküle der in der Luft vorhandenen Gase werden von einer Klangquelle in Bewegung versetzt, so dass Wellen von stärkerem und schwächerem Druck entstehen (vgl. Abbildung 1). Diese Wellen versetzen das Trommelfell in Schwingung, die im Innenohr in Nervenströme umgesetzt wird, welche das Gehirn auswerten kann.[8]

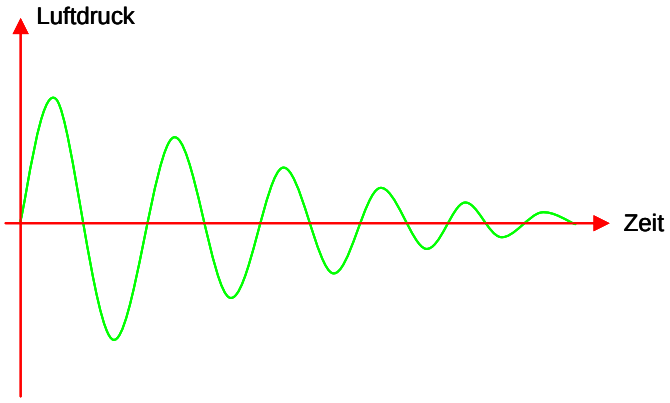

Abbildung 1: Analoges Audiosignal

Nach einem ähnlichen Prinzip funktionieren auch sämtliche analogen Klangerzeuger und -Rezeptoren (z.B. Lautsprecher und Mikrophone): Klangerzeuger erzeugen Schwingungen, die – meistens durch die Luft – zum Klangrezeptor gelangen. Dieser wandelt sie über Induktion oder Kapazität in elektrische Ströme um. In der analogen Audiotechnik werden diese Ströme dann über Audiokabel für eine eventuelle Weiterverarbeitung übertragen oder auf analogen Speichermedien (Kassette, Tonband) gespeichert. An keinem Punkt findet eine Kodierung dieser Ströme in ein digitales Format statt. Dementsprechend ist die Qualität der Übertragung in der analogen Audiotechnik abhängig von der Qualität und den Rauschabständen[9] der verwendeten Übertragungs- und Speichermedien.

---

7 Der Einfachheit halber wird hier vom Übertragungsmedium Luft ausgegangen. Für alle anderen Stoffe und Stoffzusammensetzungen gelten diese Erkenntnisse analog.

8 Vgl. [WATKINSON], S. 32ff.

9 Der Signal-Rauschabstand bezeichnet das Verhältnis von Nutzsignal zu Störsignal. ([NET-LEXIKON])

1.2. Digitalisierung

„The conversion of audio to the digital domain allows tremendous opportunities which were denied to analog signals.“[10]

Die digitale Audiotechnik fängt dort an, wo Klänge entweder von digitalen Medien (z.B. Computer) erzeugt werden, um später über einen D/A-Wandler[11] ausgegeben zu werden, oder wo Klänge aus der analogen Welt über A/D-Wandler[12] in ein digitales Format übersetzt werden.

Abstrakt gesehen kann ein A/D- oder D/A-Wandler in zahlreichen Varianten auftreten. So kann das Einspielen einer Melodie über ein Keyboard in den Computer auch als A/D-Wandlung verstanden werden. Ist allerdings in der Fachliteratur oder in dieser Arbeit von A/D- oder D/A-Wandlern die Rede, so sind damit die für das Abtasten kontinuierlicher analoger Signale bzw. für das Erzeugen analoger Signale aus Abtastwerten benötigten Geräte gemeint.

Der Digitalisierungsprozess von analogen Klängen kann in zwei Schritte unterteilt werden: Zunächst wird das theoretisch unendlich genaue Eingangssignal in Werte von endlicher Genauigkeit umgewandelt (sog. *Quantisierung*). Dieser Schritt ist zwangsläufig mit einem Informationsverlust verbunden (von unendlicher in endliche Auflösung). Allerdings kann dieser Informationsverlust mit vertretbarem Aufwand so gering gehalten werden, dass er jenseits des vom menschlichen Ohr wahrnehmbaren Bereichs liegt. Im zweiten Schritt werden die bei der Quantisierung gewonnenen Werte in ein digitales Format umgewandelt (sog. *Kodierung*).

Prinzipiell unterscheidet man zwei grundlegend verschiedene Arten der digitalen Haltung von Klangdaten: Der am weitesten verbreitete Ansatz ist das sog. *Sampling*, bei dem versucht wird, einen Klang anhand regelmäßiger Abtastwerte des analogen Signals so detailgetreu wie möglich abzubilden. Im Gegensatz dazu steht der – auf musikalisches Material beschränkte – Ansatz der Beschreibung des Klangs durch eine Reihe von Musiknoten und anderer sog. *Events* (ähnlich einer Partitur), der im MIDI-Format Verwendung findet. Der Name *MIDI* hat sich auch als gängigste Bezeichnung für diesen Ansatz der Digitalisierung von Klangmaterial durchgesetzt.

1.2.1 Sampling

Der Quantisierungsprozess läuft beim Sampling folgendermaßen ab: Der Wandler tastet in regelmäßigen Abständen den aktuellen Wert des analogen Signals ab und wandelt ihn in einen diskreten Wert um. Dieser Wert wird als *Sample* (engl. für Stichprobe) bezeichnet, daher der Name *Sampling*. Die Genauigkeit der Abtastung eines Samples wird durch die Samplegröße bestimmt. Sie bestimmt die Auflösung, mit der die einzelnen Samples dem abgetasteten analogen Wert angenähert werden. Je mehr mögliche Werte es pro Sample gibt, desto näher ist das Ergebnis dem analogen Original. Gängige Größe hierfür ist eine Samplegröße von 16 Bit (Audio-CD-Standard), also $2^{16} = 65536$ mögliche Werte, die ein Sample annehmen kann. Für reine Sprachaufnahmen werden oft 8 Bit große Samples verwendet, wohingegen im Studioumfeld inzwischen oft mit 24 oder 32 Bit gearbeitet wird. Bei zu geringer Auflösung erzeugt der Samplingvorgang ein so genanntes Quantisierungsrauschen. Dieses Rauschen ist signalabhängig und wird daher als störender empfunden, als ein kon-

---

10 Aus [WATKINSON], S. 8.
11 Digital/Analog-Wandler
12 Analog/Digital-Wandler

stantes Hintergrundrauschen. Ist das Eingangssignal nicht gut ausgesteuert, also relativ leise, verringert sich die effektive Auflösung, da die Anzahl der möglichen Werte, die das Signal annehmen kann, sinkt.[13]

Die Häufigkeit der Abtastung ist der zweite Faktor, der darüber entscheidet, wie originalgetreu das Ergebnis ist. Sie wird als *Sampling-Frequenz* oder *Samplingrate* bezeichnet und in Hertz (Hz)[14] angegeben. Da das menschliche Ohr bestenfalls Frequenzen im Bereich bis zu 20 kHz wahrnehmen kann und nach dem Abtasttheorem von Shannon/Nyquist[15] die Abtastung eines analogen Signals mit mehr als der doppelten höchsten aufzuzeichnenden Frequenz erfolgen sollte, hat sich für gute Audioqualität die Sampling-Frequenz 44.1 kHz (Audio-CD-Standard) etabliert. Weitere gängige Frequenzen sind unter anderem 11.025 Hz, 22.050 Hz und 48 kHz.

Im Studiobereich wird inzwischen aber auch dort, wo es die Soft- und Hardware unterstützt, hin und wieder mit deutlich höheren Sampling-Frequenzen (96 oder gar 192 kHz) gearbeitet. Solch hohe Sampling-Frequenzen führen zu einem starken Anstieg des Datenaufkommens und sollten daher nur dort eingesetzt werden, wo sie auch benötigt werden, da keine für das menschliche Ohr verwertbaren Mehrinformationen darin enthalten sind[16]. Sinnvoll kann der Einsatz hoher Sampling-Frequenzen dann sein, wenn Rundungsfehler bei späteren Konvertierungen in andere Sampling-Frequenzen minimiert werden sollen oder zur Verringerung des Aliasingeffekts, die jedoch effizienter durch den Einsatz guter Lowpassfilter vor dem A/D-Wandler erreicht werden kann.[17]

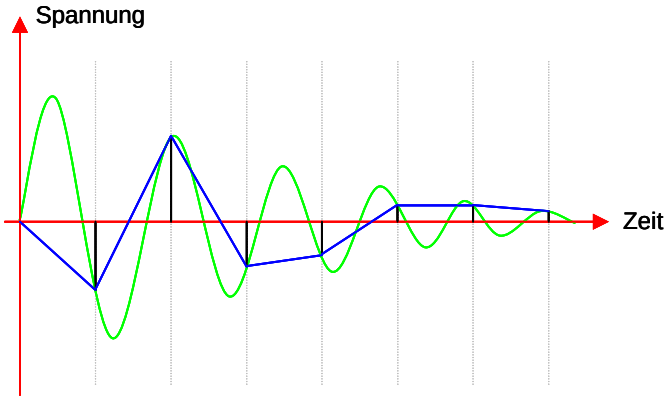

Abbildung 2: Visualisierung des Aliasingeffekts beim Abtasten

Aliasing beim Abtasten bezeichnet den Effekt, dass Frequenzen, die höher als die halbe Samplingrate sind, im abgetasteten Signal als tiefere Frequenzen wieder auftauchen und somit zu einer hörbaren Klangverfälschung führen.[18] Abbildung 2 zeigt diesen Effekt: Das blaue Signal als grobes Ergebnis der Abtastung weist eine niedrigere Frequenz auf, als das grüne Originalsignal.

---

13 Vgl. [BRÜSE], S. 16ff.

14 1 Hz steht für eine Abtastung pro Sekunde, $1\,Hz = \frac{1}{s}$.

15 Vgl. [NET-LEXIKON].

16 Vgl. [WATKINSON], S. 729f.

17 Vgl. [BRÜSE], S. 11ff.

18 Vgl. [BRÜSE], S. 14ff.

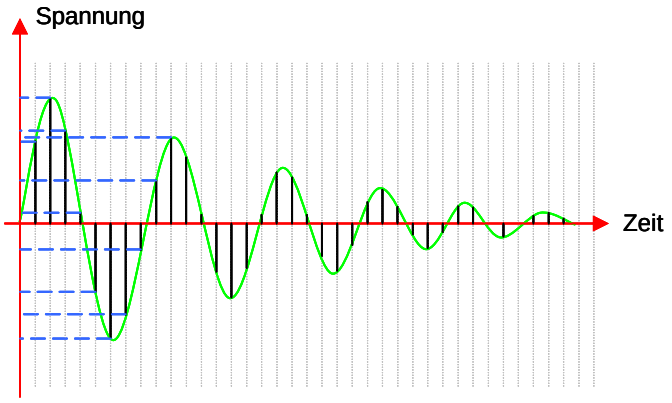

Abbildung 3: Zeitliche Quantisierung beim Sampling

Das analoge Signal wird also beim Sampling-Vorgang zweimal quantisiert, zunächst in ein zeitliches Raster (vgl. Abbildung 3) und anschließend in ein Raster möglicher Werte pro Abtastung (vgl. Abbildung 4).

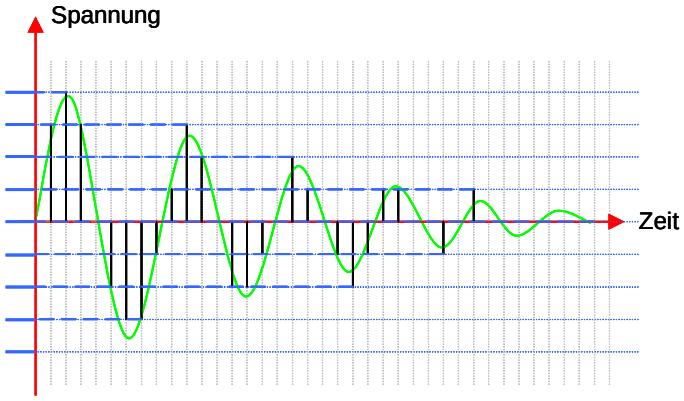

Abbildung 4: Quantisierung der Abtastwerte beim Sampling

Für die Abbildungen wurde in beide Richtungen zur Veranschaulichung ein sehr grobes Raster gewählt. Wenn man die so gewonnenen Werte mit dem analogen Originalsignal vergleicht, wird der Informationsverlust deutlich (vgl. Abbildung 5).

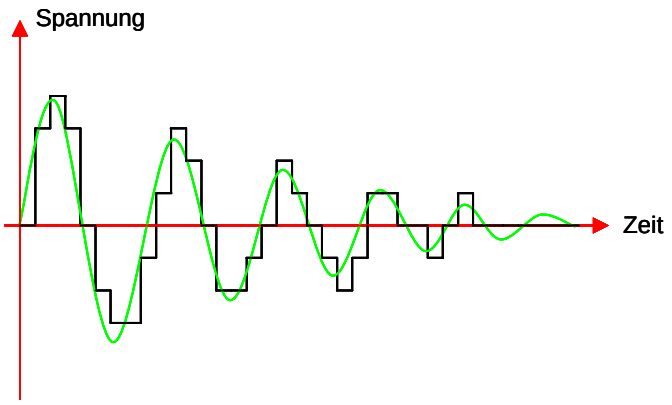

Abbildung 5: Ergebnis der Digitalisierung im Vergleich zum Originalsignal

Entscheidend für die Qualität des Ergebnisses ist zudem die Synchronisation der Abtastvorgänge. Man kann sich leicht vorstellen, dass es wenig sinnvoll ist, wenn man 44.100 Mal pro Sekunde abtastet, diese Abtastungen aber so unregelmäßig erfolgen, dass sie das Ergebnis verzerren. Der Prozess des Abtastens ist also hoch echtzeitsensibel. Beim Übergang von der Hardware- zur Softwareschicht werden daher Puffermechanismen eingesetzt, damit Schwankungen im Software-Timing ausgeglichen werden können.

Neben diesen Faktoren, die bei der Quantisierung eine Rolle spielen, gibt es verschiedene Wege der Kodierung der quantisierten Daten zur Speicherung oder Übertragung auf digitale Medien. Diese werden in Abschnitt 1.2.4.1 behandelt.

1.2.2 Musiknoten als digitale Klangdaten (MIDI)

Während das Sampling erst mit dem Aufkommen digitaler Technik möglich wurde, gab es eine andere Form der Quantisierung von Klangmaterial schon viel früher: Bereits aus dem 10. Jahrhundert sind erste Quellen überliefert, die Versuche belegen, Musik durch bestimmte Zeichen schriftlich festzuhalten[19]. Aus diesen Versuchen ging die Notenschrift hervor, die heute fast überall auf der Welt verwendet wird, um Musikkompositionen niederzuschreiben.

1983 wurde von einem Zusammenschluss verschiedener führender Keyboardhersteller der MIDI-Standard ins Leben gerufen, ursprünglich mit dem Ziel, von einem sog. *Masterkeyboard* aus andere Keyboards fernsteuern zu können. Hierfür wurde – vereinfacht ausgedrückt – die Notenschrift in eine digitale Form gebracht und um einige Elemente erweitert.

Der MIDI-Standard[20] definiert folgende Elemente:

- Ein Nachrichtenformat zur Kommunikation zwischen elektronischen Musikinstrumenten
- Ein Dateiformat zum Abspeichern von Sequenzen dieser Nachrichten
- Kabel, Stecker und eine digitale Form der Kodierung für die physikalische Übertragung dieser Nachrichten

---

19 Vgl. [MUSICLINE].

20 Mehr Informationen zum MIDI-Standard unter [MIDI].

Dieser Standard verschaffte dann auch sehr schnell den ersten Musikcomputern Einzug in die Studios; denn die Möglichkeit, den Gerätepark über Sequenzer- und Editorsoftware fernsteuern zu können, eröffnete bis dahin undenkbare Anwendungsszenarien. Außerdem konnten die relativ schlanken MIDI-Daten im Gegensatz zu samplingbasierten Daten auch von damaligen Prozessoren in Echtzeit verarbeitet und von damaligen Speichermedien gehalten werden. Die Computer der Atari ST Serie wurden beispielsweise serienmäßig mit MIDI-Schnittstelle ausgeliefert, was ihnen lange Zeit eine besondere Stellung auf dem noch jungen Musikcomputermarkt einbrachte (siehe 2.1).

Das MIDI-Nachrichtenformat umfasst folgende Nachrichtentypen:

- Notennachrichten (Note-On: Taste gedrückt, Note-Off: Taste losgelassen)
- Controller-Nachrichten zur Steuerung der Geräteparameter (z.B. Lautstärke, Stereoposition, Brillanz)
- Synchronisationsnachrichten (z.B. Tempo, Start, Stop), sog. MIDI-Clock
- Pitch-Bend (Zusätzliche Beeinflussung der Tonhöhe)
- Aftertouch (Druckvariation nach dem Drücken der Taste)
- Systemexklusive Nachrichten (Ein Rahmenformat, in das jeder Hersteller gerätespezifische Nachrichten packen kann)

Zur Speicherung von MIDI-Daten definiert der MIDI-Standard drei Typen von MIDI-Dateien, die in Abschnitt 1.2.4.2 genauer behandelt werden.

1.2.3 Vereinbarkeit von Sampling und MIDI

So verschieden die Ansätze von Sampling und MIDI sind, so gab es doch immer wieder Versuche, beide miteinander zu vereinen und so von den Vorteilen beider Ansätze zu profitieren. Diese Versuche gehen allesamt den Weg, dass einzelne Klangsamples mit ihren Steuerdaten zusammengefasst werden. Beim Abspielen erzeugt der Player aus diesen Informationen den gewünschten Klang. So erhält man sich die volle Kontrolle über den Klang, kann das Datenaufkommen jedoch in Grenzen halten und behält darüber hinaus die Möglichkeit, relativ einfach Änderungen an der Komposition vorzunehmen.

Nahezu alle neueren Software-Sequenzer arbeiten heute nach diesem Prinzip. Sie bieten neben der Möglichkeit, externe oder softwareemulierte Geräte über MIDI anzusteuern, auch die Option, Sampling-Datenströme zum MIDI-Datenstrom zu synchronisieren.

1.2.4 Digitale Audioformate

1.2.4.1 Samplingformate

Grundsätzlich unterscheidet man bei samplingbasierten Audioformaten zwischen komprimierten und unkomprimierten Formaten. Komprimierte Formate, wie beispielsweise das weit verbreitete MP3-Format, versuchen, den typischerweise relativ hohen Speicherbedarf samplingbasierter Audiodaten zu verringern. Dies kann zum einen dadurch erreicht werden, dass Informationen, die für das menschliche Ohr weniger wichtig sind, weniger genau abgebildet werden als solche, die für das

Hörempfinden entscheidend sind. Hier fließen psychoakustische Merkmale ein, auf die in dieser Arbeit nicht näher eingegangen werden soll. Einen anderen Weg zur Datenreduzierung beschreiben mathematische Kompressionsverfahren, die durch eine andere Anordnung der Daten die Datenmenge verringern, ohne Informationsverlust in Kauf zu nehmen.

Komprimierte Audioformate spielen in Studioumgebungen nur eine untergeordnete Rolle. Die Gründe hierfür sind zum einen, dass Speicherplatz meist nicht der begrenzende Faktor ist und der Informationsverlust aufgrund höherer Qualitätsansprüche so gering wie möglich gehalten werden soll. Zum anderen müssen die Daten vor der Verarbeitung eventuell dekodiert und nach der Verarbeitung stets wieder kodiert werden, was unnötig viel Rechenzeit kostet. Im Folgenden wird daher nur auf die Eigenschaften unkomprimierter Formate eingegangen.

In unkomprimierten linearen[21] Formaten werden die Audiodaten PCM[22]-kodiert, das heißt pro Abtastung und Kanal wird ein Samplewert gespeichert.[23] Die einzelnen Werte werden z.B. als Integer- oder Gleitkommawerte in einen fortlaufenden Datenstrom geschrieben. Die wichtigsten Formate hierfür sind das Microsoft WAVE- und das Apple AIFF[24]-Format[25]. Beide Formate ähneln einander sehr und erlauben Mono-, Stereo- oder Mehrkanaldateien sowie unterschiedliche Abtastfrequenzen und Samplegrößen. Ebenso erlauben beide das Einfügen frei definierbarer sog. *Chunks*[26], um beliebige Zusatzinformationen mit den Audiodaten speichern zu können. Von dieser Möglichkeit machen viele Anwendungen Gebrauch, um ein gängiges Format zu nutzen, aber eigene Zusatzdaten wie beispielsweise Looppunkte hinzufügen zu können.

1.2.4.2 MIDI-Format

Der MIDI-Standard beinhaltet auch die Definition eines MIDI-Dateiformats. Dabei unterscheidet man MIDI-Dateien von Typ 0, Typ 1 und Typ 2. Typ 0 umfasst nur die Daten einer Spur, also für ein einzelnes Instrument. Typ 1 stellt eine Zusammenfassung mehrerer paralleler Spuren dar, wohingegen Typ 2 mehrere Spuren erlaubt, die aber nicht unbedingt zueinander gehören müssen. Der Typ der MIDI-Datei wird im Dateikopf festgelegt.

Eine MIDI-Datei allein sagt noch relativ wenig über den tatsächlichen Klang aus. Sie enthält lediglich Daten wie beispielsweise die folgenden, die jeweils mit einem Zeitstempel versehen sind:

- Kanal 1: Programmwechsel zu Bank 4, Instrument 3
- Kanal 14: Controller 53, Wert 93
- Kanal 5: Taste C3 mit Anschlagstärke 65 gedrückt

Diese Daten allein haben noch wenig Aussagekraft, da niemand wissen kann, welches Instrument an der Stelle Bank 4, Instrument 3 zu hören sein oder welchen Einfluss Controller 53 auf den Klang haben soll. Um diesem Umstand entgegenzutreten wurde 1991 von der MIDI Manufactu-

---

21 Genau genommen können Kodierungen wie *µLAW* oder *aLAW* auch als unkomprimiert bezeichnet werden, da sie die Daten lediglich auf einer logarithmischen Skala abbilden. Hier sind aber nur lineare Kodierungen gemeint.

22 Pulse Code Modulation

23 Vgl. [WATKINSON], S. 3.

24 Audio Interchange File Format

25 Beide Formate können auch nicht-PCM-kodierte Daten enthalten; der Kodierungstyp wird im Dateikopf mit angegeben. Am häufigsten werden sie jedoch für PCM-kodierte Daten eingesetzt.

26 Der Begriff Chunk (engl. für „Stück“) steht allgemein für ein Datenpaket.

rer's Association (MMA) der General MIDI Standard (GM) definiert, der u.a. 128 Instrumentennummern jeweils fest einem bekannten Instrument zuordnet (z.B. Instrument 1: Acoustic Grand Piano) und einigen Controllernummern Bedeutung gibt.[27] Inzwischen wurde dieser Standard in GM2 erweitert. Ebenso wurden abgespeckte Versionen für den Einsatz auf mobilen Endgeräten veröffentlicht. SP-MIDI[28] konnte sich hierunter am meisten durchsetzen.

Heutzutage unterstützen die meisten Synthesizer[29] und Betriebssysteme GM. Es ist also möglich, MIDI-Daten im GM-Format beispielsweise auf Webseiten oder in Spielen einzusetzen und damit bei den meisten Anwendern ein mehr oder weniger identisches Klangerlebnis hervorzurufen (abhängig natürlich davon, wie z.B. das Acoustic Grand Piano auf unterschiedlichen Synthesizern klingt).

MIDI wird daher oft mit GM verwechselt. Im Studiobereich behält MIDI allerdings seine ursprüngliche Bedeutung. Es ist nur für die reinen Steuerinformationen zuständig. Wie welches Instrument heißt und klingt, wird außerhalb der MIDI-Daten definiert.

1.2.4.3 Strukturiertes Audio

Bisher gab es einige Formate, in denen samplingbasierte Audiodaten zusammen mit ihren Steuerinformationen gespeichert werden konnten, um so die Vorteile von Sampling und MIDI zu kombinieren. Als Beispiele wären hier die Formate MOD, RMF[30] oder XMF[31] zu nennen, die allerdings mehr für Spiele oder Handyklingeltöne eingesetzt werden und in Studioumgebungen bisher bedeutungslos blieben. Dagegen finden sich dort meist propietäre Formate der unterschiedlichen Softwarehersteller, die diese Daten in Projekten oder Songs zusammenfassen.

Einen ganz neuen Ansatz beschreibt das MPEG[32]-4 Format mit *MPEG-4 Structured Audio*: Hierin wird versucht, in einem objektorientierten Verfahren jeden Klang in seiner Charakteristik so zu beschreiben, dass er beim Abspielen von einem Synthesizer möglichst authentisch generiert werden kann. Es werden also Informationen über das verwendete Syntheseverfahren und die dazugehörigen Parameter gespeichert. Das Format für diese Musiksynthese nennt sich SAOL[33]. Hinzu kommt mit SASL[34] quasi ein MIDI-Ersatz, der Steuerdaten für musikalische Anwendungen beschreibt. Beim Abspielen erfordert dieses Verfahren natürlich eine wesentlich höhere Rechenleistung, da der Klang erst zur Laufzeit mit teilweise sehr komplexen Syntheseverfahren generiert wird, was in Zukunft aber aufgrund steigender Prozessorleistung nicht das größte Problem darstellen dürfte.

Dieser neue Ansatz, der auch parallel von einigen anderen Technologien verfolgt wird, könnte in der Zukunft eine große Rolle spielen und sich in verschiedenen Bereichen durchsetzen. Studioumgebungen sollten zumindest darauf vorbereitet sein und diese Formate kennen.

---

27 Mehr Informationen zu General MIDI: http://www.midi.org/about-midi/gm/gminfo.shtml
28 Scalable Polyphony MIDI
29 Gängige Soundkarten sind heutzutage mit einem Synthesizer-Chip zur Klangerzeugung ausgestattet.
30 Rich Media Format
31 eXtensible Music Format
32 Moving Pictures Expert Group
33 Structured Audio Orchestra Language
34 Structured Audio Score Language

1.3. Digitale Klangverarbeitung

Hat ein Audiosignal erst einmal den Weg in ein digitales System gefunden, eröffnet die heutige DSP[35]-Technik mannigfaltige Möglichkeiten. Von der Analyse über die Optimierung bis hin zur totalen Verfremdung des Ausgangssignals ist mit digitaler Technik all das komfortabler möglich, was früher mit analoger Audiotechnik bewerkstelligt werden konnte. Außerdem stehen zusätzliche Möglichkeiten offen, die mit analoger Technik undenkbar gewesen wären und das zudem um ein vielfaches schneller und billiger.

Mit Digitaler Klangverarbeitung ist meist die Verarbeitung samplingbasierter Daten gemeint. Die Verarbeitung von MIDI-Daten stellt keine besondere Herausforderung dar, da sie bereits in einem gut strukturierten und damit leicht les- und modifizierbaren Format vorliegen. Einzige Herausforderung ist hier der Echtzeitanspruch von MIDI.

Folgende Aufgaben können im Bereich der Verarbeitung von samplingbasiertem Audiomaterial angetroffen werden:

- Formatkonvertierung
- Analyse
- Klangbearbeitung (Echtzeit oder nicht)
- Abspielen und Aufnehmen

Im Folgenden werden die dafür benötigten Hard- und Softwarekonzepte vorgestellt.

1.3.1 Sound-Hardware

1.3.1.1 Soundkarten

Soundkarten findet man heutzutage in vielfachen Ausfertigungen. Im Prinzip ist eine Soundkarte eine Ansammlung von A/D- und D/A-Wandlern sowie einem Chip, der vom Rechner aus über Befehle angesteuert werden kann. Der Hersteller der Soundkarte oder – sofern der Befehlssatz der Karte bekannt ist – Drittprogrammierer können für die unterschiedlichen Betriebssysteme und Treiberschnittstellen Treiber bereitstellen, um die Soundkarte über eine definierte Schnittstelle ansprechen zu können.

Der Name Soundkarte stammt aus einer Zeit, in der diese Funktionalität tatsächlich immer auf einer ISA[36]- oder PCI[37]-Karte untergebracht wurde. Inzwischen gibt es sowohl in das Mainboard integrierte Onboard-Lösungen, als auch externe Geräte mit umfangreichen Anschlussmöglichkeiten und interner zusätzlicher Hardware.

---

35 Digital Signal Processing
36 Integrated System Architecture
37 Peripheral Component Interconnect

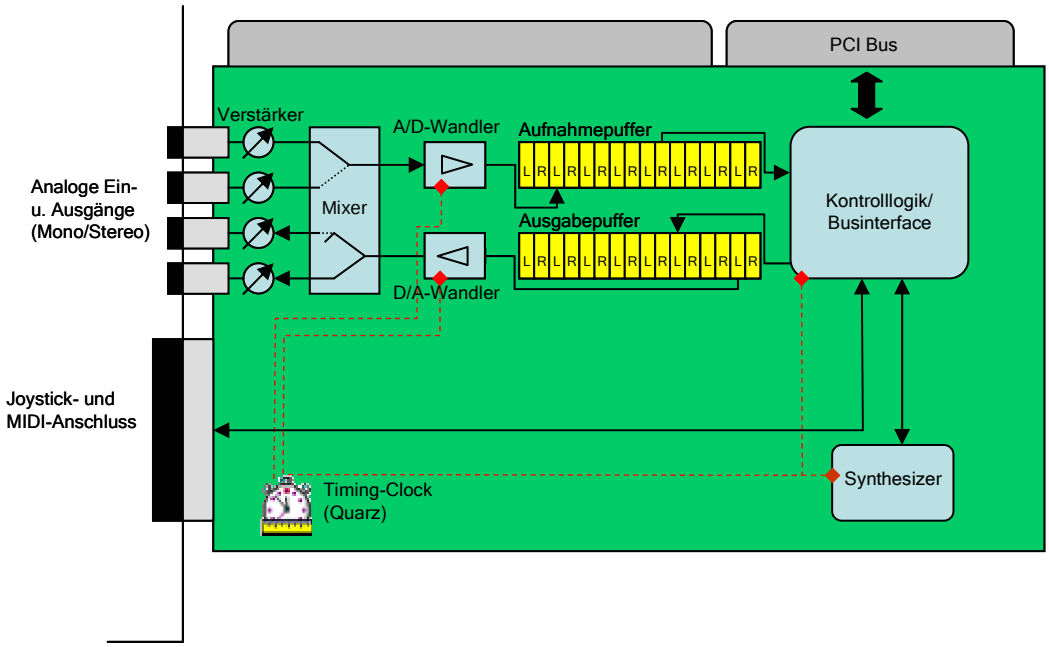

Abbildung 6: Vereinfachter schematischer Aufbau einer herkömmlichen Stereosoundkarte

Eine Soundkarte besteht mindestens aus den folgenden Bestandteilen (vgl. Abbildung 6)[38]:

Anschlüsse

Gängige Soundkarten bieten meist einen analogen Line- (Stereo) und einen Mikrofoneingang (Mono) sowie ein bis zwei analoge Ausgänge (Stereo). Hinzu kommt der Joystick-Anschluss, über den mit Hilfe eines speziellen Adapterkabels externe MIDI-Hardware angeschlossen werden kann.

Soundkarten für den Profi-Bereich bieten in der Regel eine höhere Zahl an Anschlussmöglichkeiten, darunter meist auch digitale Ein- und Ausgänge in unterschiedlichen Formaten.

Wandler

Die A/D-Wandler übersetzen analoge Eingangssignale in eine digitale Form, damit sie vom Chip der Soundkarte weiterverarbeitet werden können. Analog dazu wandeln D/A-Wandler die digitalen Daten aus dem Rechner in analoge Spannungswerte für die Ausgangssignale (vgl. 1.2.1).

Die Wandler beziehen von der Soundkarten-Clock ihre Sampling-Frequenz und lesen direkt aus dem bzw. schreiben direkt in den Hardwarepuffer. Sie bestimmen, welche Samplingraten die Soundkarte unterstützt und in welcher Auflösung (Samplegröße) die Abtastung erfolgt. Gute Wandler arbeiten mit etwas größerer Auflösung als nominell angegeben (z.B. 20 Bit statt 16 Bit) und reduzieren diese anschließend durch ein spezielles Verfahren (sog. *noise shaping*). So kann das Quantisierungsrauschen beim Sampling reduziert werden.

Hardwarepuffer

Um die echtzeitsensiblen Wandlungsvorgänge unabhängig von Timingschwankungen der Softwareschicht zu halten, wird zwischen Wandler und Chip je ein Puffer eingebaut, auf den Wandler und Chip jeweils mit einem internen Zeiger zugreifen, der nach jedem Sample um eins weiter gerückt

[38] Vgl. [BRÜSE], S. 68ff.

wird, bis er am Ende des Puffers angelangt ist und wieder an den Anfang gesetzt wird. Dabei arbeiten die Wandler mit ihrem vorgegebenen Takt, unabhängig davon, ob der Chip – also letztendlich die Software – rechtzeitig Daten im Puffer bereitgestellt bzw. aus dem Puffer ausgelesen hat. Schafft die Software das nicht in der ihr zur Verfügung stehenden Zeit, treten ungewollte akustische Effekte wie Knacksen oder Wiederholungen auf (buffer underrun oder overflow).

Meist sind die Puffer so ausgelegt, dass sie abwechselnd mit je einem Sample für den linken und einem Sample für den rechten Kanal gefüllt werden[39].

Clock

Jede Soundkarte besitzt einen eigenen Taktgeber für die Abtastvorgänge der Wandler, der meist in Form eines sehr genauen Quarzes eingebaut ist. Auch der Chip muss zu dieser Clock synchronisiert werden, um der Software diese Zeitbasis weitergeben zu können. Da unterschiedliche Soundkartenclocks nie hundertprozentig synchron sind, laufen Signale, die auf unterschiedlichen Soundkarten parallel abgespielt oder aufgenommen werden, früher oder später auseinander. Um dies zu verhindern, bieten professionelle Soundkarten die Option, ihre Clock von einem externen Zeitgeber zu beziehen oder die eigene Clock an andere Geräte zu übermitteln (vgl. 1.3.1.2).

Chip

Der Soundkartenchip bildet schließlich die Kontrolleinheit der Soundkarte. Er hat einen definierten Befehlssatz, den der Soundkartentreiber ansprechen kann, um Hardwarefunktionen auszulösen. Viele der zuvor beschriebenen Elemente befinden sich faktisch im Chip. In der Abbildung wurden sie jedoch als eigene Komponenten dargestellt.

Optionale Komponenten

Darüber hinaus gibt es viele Elemente wie GM-Synthesizer, Effektprozessoren, Surroundprozessoren, DSP-Einheiten u.v.m., die man auf einigen Karten vorfindet, auf anderen wiederum nicht. Oft werden diese bei Nichtvorhandensein durch die Treibersoftware emuliert, weshalb der Benutzer oft nicht merkt, welche Komponenten davon auf seiner Soundkarte tatsächlich vorhanden sind.

Qualitativ können sich Soundkarten in folgenden Merkmalen unterscheiden:

- Anzahl der Kanäle
- Art der Anschlüsse: analog, digital, (un-)symmetrisch, Clock-Anschlüsse
- Unterstützte Samplingraten
- Zusätzliche DSP-Prozessoren
- Qualität der Wandler
- Synchronisationsmöglichkeiten
- Qualität und Architektur der angebotenen Treiber

---

[39] Diese Art der Haltung von Mehrkanaldaten wird auch als *interleaved* bezeichnet.

1.3.1.2 Synchronisation

Überall dort, wo mehrere Hardwarekomponenten im Einsatz sind, die mit einer internen Timing-Clock arbeiten, ergibt sich das Problem, dass diese nie ganz genau im selben Takt laufen und dass so nach einiger Zeit ein Auseinanderdriften der verschiedenen Hardwarekomponenten festzustellen ist. Im Audiobereich tritt dies auf, wenn mehrere Soundkarten parallel betrieben werden oder wenn externe digitale Hardware (beispielsweise ein digitales Mischpult) eingesetzt wird. Um dieses Auseinanderdriften zu verhindern ist es nötig, festzulegen, welche Clock die Referenz für alle anderen darstellen soll, und die anderen Komponenten entsprechend zu dieser sog. *Master*-Clock zu synchronisieren. Nicht alle Soundkarten unterstützen diese Form der Synchronisation.

Um diese Synchronisation zu ermöglichen, muss der Master in der Lage sein, ein sog. *Wordclock*-Signal zu übertragen, ein Signal, das in der Frequenz des Taktgenerators oszilliert. Wordclock kann entweder über digitale Datenverbindungen zusammen mit dem Nutzsignal oder mittels einer eigenen Wordclock-Leitung übertragen werden. Daher bieten professionelle digitale Geräte wie digitale Mischpulte oder HD-Recorder stets Wordclock-Anschlüsse. Die zu synchronisierenden Geräte müssen in den sog. Slave-Modus versetzt werden und verwenden dann den Takt aus dem Wordclock-Signal als Referenz für ihre Wandler und sonstigen Funktionen[40].

Die Wordclock-Synchronisation sollte nicht mit absoluten Synchronisationsmethoden wie MIDI Time Code (MTC) oder SMPTE[41] Timecode verwechselt werden. Diese dienen vielmehr der Synchronisation unterschiedlicher Medientypen wie Audio/Video/MIDI zueinander.

1.3.2 Softwarekonzepte und -besonderheiten

Im Bereich der Programmierung von Audioanwendungen gibt es häufig wiederkehrende Problemstellungen und Konzepte, um bestimmte Anforderungen zu erreichen. Die wichtigsten werden im Folgenden vorgestellt.

1.3.2.1 Echtzeit in der Audioprogrammierung

In der DIN[42]-Norm 44330 wird Echtzeitbetrieb folgendermaßen definiert:

> „Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen."

Bei Echtzeitanforderungen unterscheidet man *harte Echtzeit* und *weiche Echtzeit*. Eine harte Echtzeitanforderung liegt dann vor, wenn die Zeitvorgaben unter keinen Umständen überschritten werden dürfen. Bestes Beispiel für harte Echtzeitanforderungen sind computergestützte Kfz-Fahrhilfen; denn ein Anti-Blockiersystem, das auch nur ein bisschen zu spät reagiert, ist genauso gut wie gar keines. Dagegen toleriert eine weiche Echtzeitanforderung geringe Überschreitungen der Zeitvorgaben, wenn sie in der Regel eingehalten werden.

---

40 Vgl. [BRÜSE], S. 139ff.

41 Society of Motion Picture and Television Engineers

42 Deutsches Institut für Normung

Im Bereich der Audioprogrammierung wird Echtzeit insbesondere an zwei Stellen gefordert: Zum einen, wenn es darum geht, samplingbasierte Audiodaten wiederzugeben oder aufzunehmen. Um beispielsweise bei der Wiedergabe sicherzustellen, dass der Puffer der Soundkarte zu jeder Zeit soweit mit Daten gefüllt ist, dass der Klang zur rechten Zeit ausgegeben werden kann, also um buffer underruns zu verhindern, muss die Quelle die Audiodaten regelmäßig innerhalb eines vorgegebenen Zeitintervalls liefern.

Sind die Daten schon vor dem Abspielvorgang bekannt, kann diese Anforderung sehr einfach durch ein Lesen der Daten im Voraus und das Verwenden eines großen Puffers erfüllt werden. In diesem Fall kann sich die Audioquelle leichte Timingschwankungen erlauben, da genug im Voraus gelesen wurde. Es handelt sich also um eine weiche Echtzeitanforderung.

Werden die Daten dagegen erst zur Laufzeit generiert und können sich bis kurz vor dem Moment des Abspielens noch ändern, was in interaktiven Applikationen die Regel ist (siehe 1.3.2.3, Echtzeitberechnung), ergibt sich ein Spannungsfeld, da die Daten erst so spät wie möglich an die Hardware geschickt werden sollen, um auf Änderungen bis in den letzten Moment reagieren zu können. In diesem Fall handelt es sich um eine harte Echtzeitanforderung, da die Audioquelle ständig in der Lage sein muss, im nächsten Zeitintervall die nötigen Audiodaten zu liefern, um buffer underruns zu verhindern. Diese machen sich meist dadurch bemerkbar, dass das Audiosignal entweder anfängt zu knacksen oder dass der Inhalt des Soundkartenpuffers wiederholt abgespielt wird. Es gilt also: Je kleiner der verwendete Puffer, desto besser die Interaktivität, aber auch desto höher die Echtzeitanforderung an die Software.

Ebenso wie bei samplingbasierten Daten spielt auch bei MIDI-Daten die Echtzeitfähigkeit der Software eine große Rolle. Um vernünftig mit einem MIDI-System arbeiten zu können, erwartet der Benutzer, dass beispielsweise MIDI-Daten, die er von einem externem MIDI-Keyboard einspielt, ohne für ihn hörbare Verzögerung an das empfangende MIDI-Gerät weitergegeben werden. Ebenso darf der Rhythmus der Noten auf keinen Fall durch hörbare Varianzen beim Ansteuern vom Sequenzer aus verfälscht werden.

Im Gegensatz zu samplingbasierten Daten werden MIDI-Daten nur dann gesendet, wenn sie auch wirklich anfallen (Beim Abspielen einer samplingbasierten Sounddatei, die nur Stille enthält, werden trotzdem laufend Daten von der Software in den Hardwarepuffer der Soundkarte geschrieben). Eine MIDI-Datei, die nur eine leere Spur enthält, würde also keinerlei Hardwareaktivität verursachen. Aus diesem Grund wird bei MIDI-Hardware in der Regel auch kein großer Puffer eingesetzt. Die Daten werden in dem Moment gesendet, in dem sie anfallen. Die Software muss also in der Lage sein, im richtigen Moment die Nachrichten an die Hardware zu schicken, und kann sie nicht schon im Voraus senden. Zwar kann die Software intern Puffer verwenden, was auch gemacht wird, allerdings ändert das nichts daran, dass MIDI-Software stets harte Echtzeitanforderungen zu erfüllen hat.

Da das MIDI-Nachrichtenformat seriell ist, also mehrere gleichzeitig anstehende Nachrichten nicht gemixt werden können, sondern nacheinander gesendet werden müssen, sind minimale Timingschwankungen unvermeidlich. Bei hohem Datenaufkommen müssen hier Konzepte angewandt werden, die diese Schwankungen nicht hörbar werden lassen, beispielsweise eine Ausdünnung redundanter Daten, das Vorziehen bestimmter Nachrichten oder eine Priorisierung wichtigerer Nachrichten, was nicht immer mit befriedigendem Ergebnis möglich ist.

Es gibt allerdings inzwischen auch Ansätze, MIDI-Interfaces mit umfangreicherer Puffer-Technik auszustatten, um das Timing zu verbessern und gleichzeitig Datenreduktion und -Priorisierung zu ermöglichen. Diese sind allerdings fast alle herstellerspezifisch und setzen spezielle Hardware voraus. Beispiele hierfür sind Emagics AMT-Technologie[43] oder Steinbergs LTB[44].

1.3.2.2 Verarbeitungsketten

Zur Verarbeitung zeitabhängiger Medien – wie es Audiodaten sind – hat sich in der Softwareentwicklung das Konzept von Verarbeitungsketten durchgesetzt.

> „Unter einer Verarbeitungskette versteht man in der Medienverarbeitung eine Kette von Stationen, die ein Eingabemedium chunkweise parallel verarbeiten. [...] Die beteiligten Stationen nennt man Transformatoren [...]. Der Datenfluss zwischen ihnen läuft über so genannte Ports ab.“[45]

Man unterscheidet Ein- und Ausgabeports, von denen jeder Transformator beliebig viele (oder keine) haben kann. Dabei spielt es keine Rolle, ob ein Transformator eine Hard- oder Softwareeinheit repräsentiert. Verarbeitungsketten bieten den Vorteil, dass die Verarbeitung auf einmal vorgenommen werden kann, indem Datenchunks vom Anfang bis zum Ende der Kette durchgereicht und dabei verarbeitet werden, wobei jeder Transformator nur seinen Vorgänger und Nachfolger kennen muss. Sind die beteiligten Transformatoren schnell genug, kann die gesamte Verarbeitung in Echtzeit ablaufen.

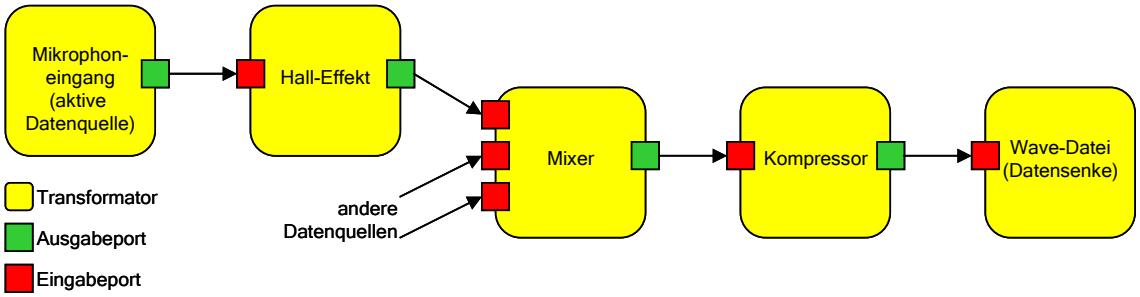

Abbildung 7: Verarbeitungskette mit aktiver Quelle (Push-Ansatz)

Eine Verarbeitungskette beginnt jeweils mit einer Datenquelle und endet mit einer Datensenke. Datenquellen können aktiv[46] oder passiv[47] sein. Je nach Art der Datenquelle unterscheidet man zwei Ansätze von Verarbeitungsketten: Beim *Push*-Ansatz (vgl. Abbildung 7) steht zu Beginn der Kette eine aktive Datenquelle, die von sich aus Datenchunks sendet. Die Chunks werden dann jeweils an den nächsten Transformator weitergegeben, bis das Ende der Kette erreicht ist, wo sie z.B. in eine Datei geschrieben werden. Beim *Pull*-Ansatz dagegen beginnt die Verarbeitung nur auf Anfrage durch die Datensenke am Ende der Kette. Diese Anfrage wird bis zur meist passiven Quelle weitergegeben, die die geforderten Daten in die Kette schickt.[48]

---

43 Active MIDI Transmission, http://www.emagic.de/products/hw/amt/amt.php?lang=DE
44 Linear Time Base, http://www.steinberg.de/ProductPage_sb.asp?Product_ID=2022&Langue_ID=4
45 Aus [EIDENBERGER], S. 43.
46 Aktive Quellen schicken von sich aus Chunks in die Verarbeitungskette, z.B. Mikrophon.
47 Passive Quellen stellen Chunks nur auf Anfrage zur Verfügung, z.B. Audiodatei.
48 Vgl. [EIDENBERGER], S. 43ff.

1.3.2.3 Nichtdestruktive Bearbeitung

Eine Herausforderung für die Programmierung von Audioanwendungen ist es, dem Benutzer möglichst umfangreiche Manipulationen des Ausgangsmaterials zu ermöglichen, dabei aber so lange wie möglich nichtdestruktiv zu arbeiten, um Änderungen nicht gleich permanent zu machen. Das bedeutet, dass die Bearbeitungen keine tatsächlichen Auswirkungen auf das Ausgangsmaterial haben – sondern beispielsweise nur auf eine Kopie –, solange vom Benutzer nicht explizit das Gegenteil gewünscht wird. Um dies zu erreichen existieren folgende Möglichkeiten, die auch kombiniert eingesetzt werden können:

Arbeiten auf Kopien

Bei dieser Methode wird automatisch zu Beginn jeder Bearbeitung eine Kopie der Ausgangsdaten angelegt. Damit ist es jederzeit möglich, zum Ausgangsmaterial zurückzuspringen, unabhängig davon, was mit den Kopien gemacht wurde. Dem Benutzer sollte zu jeder Zeit klar sein, wann er auf Kopien arbeitet und wann auf dem Original. Ein Nachteil von Kopien ist, dass der Speicherplatzbedarf sehr hoch werden kann (mindestens doppelt so hoch wie das Ausgangsmaterial).

Speichern jedes Bearbeitungsschritts

Bei diesem Ansatz wird jeder vom Benutzer durchgeführte Bearbeitungsschritt inklusive aller notwendigen Daten so gespeichert, dass eine Rückkehr zum vorigen Stand jederzeit möglich ist. Er kann sehr gut für Undo-/Redo-Funktionen eingesetzt werden. Auch hierbei kann das Datenaufkommen unter Umständen enorm groß werden.

Schnittlisten

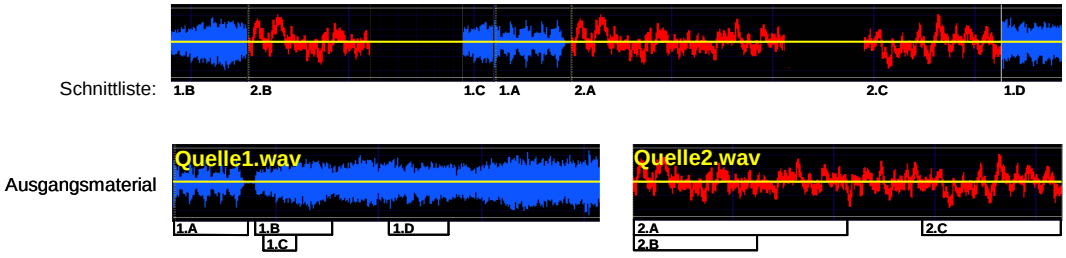

Abbildung 8: Beispiel für eine Schnittliste

Geht es um das reine zeitliche Anordnen, Löschen, Kopieren usw. von Audiomaterial, stellen Schnittlisten eine einfache Möglichkeit dar, das hohe Datenaufkommen der beiden vorherigen Ansätze zu verhindern. Schnittlisten funktionieren so, dass das Ausgangssignal nach Bedarf zerschnitten und wieder neu angeordnet wird. Sie definieren die Reihenfolge, in der die betreffenden Teile des Ausgangsmaterials aufeinander folgen[49]. Auf diese Weise muss nicht das ganze Bearbeitungsergebnis gespeichert werden, sondern lediglich die neue Abfolge. Schnittlisten können wiederum als Quellen für weitere Schnittlisten verwendet werden. Sie können zudem um elementare Grundbearbeitungen wie Umkehren einzelner Abschnitte, Änderung der Lautstärke o.ä. erweitert werden und

[49] Vgl. [BRÜSE], S. 51ff.

können viel Speicherplatz und Bearbeitungszeit sparen. Allerdings müssen die Ergebnisse ausreichend schnell beim Abspielen berechnet werden können.

Echtzeitberechnung

Ein ähnlicher Ansatz wie Schnittlisten ist die komplette Berechnung der Bearbeitungen zum Zeitpunkt des Abspielens in Echtzeit. Dies ist die intuitivste Methode für den Benutzer, da er keine Berechnungszeiten in Kauf nehmen muss, um seine Änderungen zu hören, und da er auf die Bearbeitungen noch zum Zeitpunkt des Abspielens Einfluss nehmen kann. Nur mit Echtzeitberechnung ist es möglich, das Gefühl des Arbeitens mit analoger Technik nachzuempfinden. Echtzeitberechnung erfreut sich dementsprechend hoher Beliebtheit.

Der Nachteil von Echtzeitberechnung ist die Echtzeitanforderung. Die Anzahl der simultan eingesetzten Effekte und Bearbeitungen erhöht die Prozessorlast und kann irgendwann dazu führen, dass ein Berechnen sämtlicher Bearbeitungen in Echtzeit nicht mehr möglich ist.

Für ein CPU[50]-sparendes Arbeiten mit Schnittlisten oder Echtzeitberechnung bzw. für das Speichern des Bearbeitungsergebnisses in einer neuen Datei bieten moderne Programme mit dem sog. *Bouncing* oder *Freezing* Möglichkeiten, einige Bearbeitungen einmalig in die Quelldaten hineinzurechnen, um sie dann nicht mehr in Echtzeit berechnen zu müssen. Dies geht natürlich auf Kosten der Interaktionsmöglichkeiten.

1.3.2.4 Analysen und Effekte

Eine Audioanwendung stellt dem Benutzer in der Regel eine ganze Reihe an Verarbeitungsmöglichkeiten für das Ausgangsmaterial zur Verfügung. Handelt es sich hierbei lediglich um eine Auswertung des Materials zur Gewinnung von Informationen über enthaltene Frequenzen, Lautstärke etc., so spricht man von *Analysen*. Wird dagegen das Ausgangsmaterial durch die Verarbeitung verändert, spricht man von *Effekten*, wobei Effekte in der Regel auch eine Analyse beinhalten.

Echtzeitanforderung

Man unterscheidet Verarbeitungen nach ihrer Echtzeitanforderung: Erfolgt die Verarbeitung als eigener Prozess, der vom Benutzer explizit gestartet wird, so besteht keine Echtzeitanforderung – man spricht in diesem Fall von *Offline*-Verarbeitung. Vorteil der Offline-Verarbeitung ist es, dass aufwändige Algorithmen eingesetzt werden können, da die CPU-Last nicht besonders gering gehalten werden muss. Außerdem ist es bei der Offline-Verarbeitung möglich, das Ausgangsmaterial als Ganzes zu analysieren, weshalb einige Verarbeitungen nur offline möglich sind[51].

Findet die Verarbeitung dagegen erst zur Zeit des Abspielens oder Aufnehmens in Echtzeit statt, spricht man von Echtzeitverarbeitung (Vor- und Nachteile siehe Echtzeitberechnung im vorherigen Abschnitt). Bei der Echtzeitverarbeitung können stets nur der aktuell zu verarbeitende Datenchunk sowie die bisher verarbeiteten Chunks als Grundlage für eine Analyse herangezogen werden.[52]

---

50 Central Processing Unit

51 Ein Beispiel für eine reine Offline-Verarbeitung ist das Umkehren einer Audiodatei, was als Online-Verarbeitung nicht möglich ist.

52 Vgl. [BRÜSE], S. 44f.

Analysen

Analysen können sich auf verschiedenste Aspekte beziehen. Hier sind der Fantasie keine Grenzen gesetzt. Es existieren sogar schon Ansätze, um aus einem gegebenen Audiomaterial per Analyse den Musikstil oder Interpreten zu ermitteln. In der Regel werden Analysen aber mit dem Ziel eingesetzt, das Ausgangsmaterial auf einen der folgenden Aspekte zu untersuchen:

- Lautstärke
- Lautheit (Empfundene Lautstärke)
- Frequenzverteilung
- zeitlicher Verlauf dieser Aspekte

Die Analyse der Lautstärke stellt keine Schwierigkeit dar, da PCM-kodierte Audiodaten für jedes Sample direkt den Lautstärkewert beinhalten. Eine etwas größere Herausforderung ist das Analysieren eines Signals nach seiner Frequenzverteilung, was auch für eine Lautheitsmessung nötig ist, da diese im Gegensatz zur Lautstärke abhängig von den Frequenzen ist.

Hierfür wird meistens die mathematisch relativ komplizierte Fast Fourier Transformation (FFT) verwendet[53], die auch eine gewisse CPU-Belastung mit sich bringt. FFT beruht auf der Erkenntnis, dass sich jedes periodische Signal theoretisch als Summe von Sinusschwingungen verschiedener Frequenzen und Intensitäten darstellen lässt. Um FFT auch auf nichtperiodische Signale anwenden zu können, wird ein kleiner Trick angewandt: Es wird ein bestimmter Ausschnitt des Signals betrachtet (sog. FFT-Window) und so behandelt, als würde sich dieses Signal stetig wiederholen. Das FFT-Window wird dann im Signal verschoben. Durch diesen Trick entstehen kleine Fehler, die sich aber in der Summe der Einzelbetrachtungen in der Regel ausgleichen. Als Ergebnis der FFT erhält man die Frequenzbestandteile eines Signals und deren Intensitäten.

Eine relativ neue Alternative zur FFT stellt die sog. *Wavelet*-Transformation dar. Diese teilt das Signal in nichtperiodische Anteile (sog. Wavelets) auf und eignet sich daher besser für die Analyse unperiodischer Signale, wie es Audiosignale meistens sind[54].

Auf mathematische Details der beiden Ansätze wird hier verzichtet und stattdessen auf die einschlägige Fachliteratur verwiesen: Über die FFT finden sich Abhandlungen in zahlreichen Mathematikbüchern; für Details über Wavelets, insbesondere in Bezug auf Audioprogrammierung sei auf [BÖMERS] verwiesen.

2. Audio auf verschiedenen Betriebssystemplattformen

2.1. Historische Entwicklung

Als in den 80er Jahren die Software Einzug in Studioumgebungen hielt, war die Nutzung noch ausschließlich auf den MIDI-Bereich beschränkt. Die hohen Anforderungen an Speicherkapazität

53 Vgl. [LINDLEY], S. 188ff.
54 Vgl. [STEWART].

und Prozessorgeschwindigkeit, die der – heute allgemein übliche – Einsatz samplingbasierter Daten mit sich bringt, konnten damals von keiner erschwinglichen Rechnerplattform erfüllt werden. Der Commodore C64 gilt als erster Musikcomputer, hatte aber eher experimentellen Charakter[55]. Dagegen fanden sich die Rechner der Atari ST-Serie bald auch in professionellen Studios wieder. *„[Der Atari 1040 ST] konnte sich schnell auf dem Musikermarkt etablieren. Es gibt sehr gute Programme [...], zudem verfügten die ST's als einzige Computer der 80er Jahre über eine serienmäßig eingebaute MIDI-Schnittstelle"*[56]. Wegen *„Missmanagement"*[57] konnte Atari diesen Vorsprung allerdings nicht dauerhaft halten oder gar ausbauen und wurde in den 90er Jahren von Macintosh-Systemen verdrängt. Eine Zeit lang war Apple damit unumstrittener Marktführer, bis die weite Verbreitung der Intel-PCs dazu führte, dass auch diese nach und nach in den Tonstudiomarkt drängten. Bestehende Software wurde für Windows portiert, und auch die benötigte Hardware wurde angeboten.

Ende der 90er Jahre ergab sich zwischen Microsoft und Apple ein einigermaßen ausgeglichen aufgeteilter Markt, wobei Apple-Produkten ein etwas professionelleres Image anhaftete, da Hardware und Betriebssystem aus einem Haus kommen. Das verschaffte ihnen in Profi-Studios weiterhin einen Vorsprung, wohingegen die billigeren Windows-basierten Microsoftprodukte kleinere Studios und Hobbyproduzenten erreichten.

Im Jahr 1997 entstand mit BeOS ein Betriebssystem, das speziell den Anforderungen von Multimediasoftware gerecht werden wollte. Dieses Projekt scheiterte hauptsächlich an der zögerlichen Unterstützung durch die etablierten Hard- und Softwarehersteller und an der Marktdominanz von Microsoft und Apple. Nach der Insolvenz der Firma Be, Inc. im Jahr 2002 wird BeOS als OpenBeOS[58] heute noch als Open-Source-Initiative von engagierten Programmierern weiterentwickelt. Ebenso wird von der Firma YellowTab an einer kommerziellen Version von BeOS gearbeitet[59]. Es ist nicht auszuschließen, dass BeOS in ein paar Jahren wieder auf dem Markt für Audio-Betriebssysteme zu finden sein wird; derzeit ist es dort allerdings bedeutungslos[60].

In den letzten Jahren konnte sich aber noch eine weitere Alternative zu den beiden Branchenriesen etablieren: Das Open-Source-Betriebssystem Linux konnte mit der Einführung des ALSA[61]-Treibermodells[62] einen großen Sprung in Richtung Studiotauglichkeit machen. Seine Eignung für den professionellen Audiobereich ist damit unumstritten. Zwar schreckt es momentan noch viele Benutzer ab, sich mit dem etwas schwieriger zu handhabenden Linux zu beschäftigen, und auch die branchenführenden Softwarehersteller tun sich mit einer Linuxunterstützung schwer[63]. Einige Anwendungen[64] zeigen aber, dass Linux sich schon jetzt nicht hinter den Betriebssystemen von Microsoft und Apple verstecken muss[65].

---

55 Inzwischen bietet z.B. die schwedische Firma *elektron* Synthesizer an, die auf dem original Soundchip des C64 basieren, http://www.sidstation.com/.

56 Aus [VOGT].

57 Aus [VOGT].

58 http://www.openbeos.org/

59 http://www.yellowtab.com/

60 Vgl. http://www.beosonline.de/

61 Advanced Linux Sound Architecture

62 Siehe 2.3.

63 Die GNU General Public License (GPL), unter der Linux erscheint, besagt, dass alle darauf basierenden Anwendungen ihren Quellcode offen legen müssen und dass Kopien davon kostenlos vertrieben werden dürfen, womit manche kommerzielle Softwarehersteller verständlicherweise ihre Probleme haben

64 Eine umfassende Sammlung von Audiosoftware für Linux findet sich auf http://www.linux-sound.org/.

65 Vgl. [PHILLIPS].

2.2. Heutige Bedeutung

Ein Blick auf das Softwareangebot und die Stimmung in Benutzerkreisen ergibt folgendes Bild für die heutige Bedeutung der unterschiedlichen Betriebssystemplattformen für professionelle Audiosoftware:

Apples Macintosh-Plattform konnte sich bis heute ihren professionellen Ruf erhalten und hat daher im professionellen Umfeld nach wie vor eine stärkere Position[66]. Allerdings konnte die Windows-Plattform nicht zuletzt dank der Zusammenführung des stabilen Windows NT und des multimediafreundlichen Windows 98/ME im Betriebssystem Windows XP viel Boden gut machen und ist im semiprofessionellen Umfeld und im Hobbybereich, der einen nicht zu unterschätzenden Markt darstellt, verstärkt anzutreffen. In der Bedeutung für anspruchsvolle Audiosoftware können Mac-Plattform und Windows-Plattform daher in etwa gleichgesetzt werden.

Auch Atari-Rechner finden sich noch in einigen Studios für die Erfüllung von Aufgaben im MIDI-Bereich. Allerdings wird so gut wie keine neue Software für diese Plattform entwickelt und die Systemleistung liegt auch weit hinter dem, was heutzutage möglich ist. Daher werden auch diese Ataris früher oder später durch neue Rechner ersetzt werden.

Linux konnte sich inzwischen auch auf dem Audiosektor etablieren, leidet aber derzeit noch an der mangelnden Unterstützung der Hard- und Softwarehersteller. Seine Bedeutung für Audiosoftware sollte aber gerade mit Blick in die Zukunft nicht unterschätzt werden. Das Projekt *Agnula*[67] ist beispielsweise eine Linux-Distribution speziell für Multimedia-Systeme.

BeOS spielt zwar derzeit keine Rolle auf dem Audiosoftwaremarkt, es ist jedoch nicht auszuschließen, dass es in Zukunft doch noch den Durchbruch schafft.

2.3. Audiotreibermodelle

Für Audiohardware gibt es zahlreiche Treibermodelle mit verschiedenen Ausrichtungen und für verschiedene Betriebssystemplattformen. Im Folgenden soll eine Auswahl der für professionelle Audiosoftware bedeutendsten Modelle vorgestellt werden.

Direct Sound

Direct Sound ist Teil der DirectX-Programmierbibliothek von Microsoft für die Windows-Betriebssysteme ab Windows 95, die sich dort zum Standard für die Programmierung von Multimediaanwendungen und Spielen entwickelt hat. Wie das gesamte DirectX-Framework bietet auch Direct Sound bestimmte Funktionen unabhängig davon an, ob sie von der darunterliegenden Hardware unterstützt werden[68]. Insbesondere sind das Funktionen, die gerade für Spiele brauchbar sind wie z.B. 3D-Sound; eine Optimierung für Studiotauglichkeit wird hingegen nicht angestrebt.

---

66 Im Gegensatz zu Microsoft plant Apple eigenen Angaben zufolge in Zukunft verstärkt für den Audioproduktionsmarkt. Damit wurde auch der Kauf des wichtigen Audiosoftwareherstellers *Emagic* (http://www.heise.de/newsticker/meldung/28696) im Jahr 2002 begründet.

67 http://www.agnula.org/

68 Funktionen, die die Hardware nicht bereitstellt, werden softwarebasiert emuliert und sind somit etwas langsamer.

Direct Sound ermöglicht zwar geringere Latenzen[69] als MME[70]-Treiber, allerdings keine, die professionellen Ansprüchen genügen könnten[71]. Auch ist es über Direct Sound nicht möglich, Mehrkanalhardware als solche anzusprechen. Die Soundausgabe ist nur in Mono oder Stereo vorgesehen.

Direct Sound hat daher im Studiobereich keine besondere Bedeutung, wird aber an dieser Stelle erwähnt, da Suns Java-Sound-Implementierung in der JDK Version 1.5.0 unter Windows mit Direct Sound arbeitet.

ASIO

Das ASIO[72] Treibermodell wurde von der Firma Steinberg entwickelt und im Jahr 1997 zusammen mit der Software Cubase VST eingeführt. Dabei war das Ziel, Audioapplikationen zu ermöglichen, die Audiohardware mit niedriger Latenz und mehreren gleichzeitig ansprechbaren Ein- und Ausgängen unterstützen. Damals stellte sich die Situation folgendermaßen dar:

> „The personal computer platform really misses a relatively simple way of accessing multiple audio inputs and outputs. Today's operating system's audio support is designed for stereo input and stereo output only. There is no provision to extend this without creating major problems, i.e. synchronization issues between the different input and output channels."[73]

Es gelang Steinberg, mit ASIO einen Quasi-Standard zu etablieren, der sich schnell durchsetzte. Hersteller professioneller Mehrkanalsoundkarten boten ASIO-Treiber an und für die gängigen Applikationen wurde ASIO-Unterstützung implementiert. ASIO unterstützt sowohl Windows- als auch Mac-Betriebssysteme.

ASIO hat allerdings auch den Nachteil, dass zu jeder Zeit nur ein ASIO-Treiber aktiv sein kann und dabei nur von einem Programm genutzt werden darf.

An dieser Stelle sollte das Projekt *ASIO4All*[74] Erwähnung finden. ASIO4All ist ein ASIO-Treiber, der mit sämtlichen Consumer-Soundkarten funktioniert, deren Treiber WDM[75] unterstützen. Die Hersteller dieser Soundkarten bieten üblicherweise keine ASIO-Treiber an. Der kleine Treiber, der sich zwischen WDM-Treiber und Softwareanwendung schiebt, kann natürlich aus einer Stereokarte keine Mehrkanalkarte machen, funktioniert aber sonst erstaunlich gut und ermöglicht niedrige Latenzen auch mit einfachen Onboard-Soundkarten.

EASI

Steinbergs größter Konkurrent, die Firma *Emagic*, veröffentlichte 1999 einen ähnlichen Treiber-Standard namens EASI[76], der einfacher aufgebaut und leistungsfähiger als ASIO sein sollte und zudem – im Gegensatz zu ASIO – ohne Lizenzierung offen für alle Entwickler und gut dokumentiert. EASI konnte sich aber nicht entscheidend gegenüber ASIO durchsetzen. Spätestens seit Ema-

---

69 Allgemein bezeichnet der Begriff Latenz in der Computertechnik die wahrgenommene Antwortzeit eines Geräts auf ein Signal (vgl. [NET-LEXIKON]). Im Audiobereich ist damit meist die durch eine Soundkarte und ihren Treiber – insbesondere durch Pufferung – entstehende Verzögerungszeit im Audiodatenfluss gemeint.

70 Multi Media Extension: Die Standard-Treiberschnittstelle für Audio auf Windows-Systemen.

71 Laut Microsoft entstehen mindestens 30 ms Latenz bei Verwendung von Direct Sound, vgl. [MICROSOFT].

72 Audio Streaming Input Output, 1997 entwickelt; Version 2.0 erschien 1999.

73 Aus [ASIO], S.3.

74 http://michael.tippach.bei.t-online.de/asio4all/

75 Windows Driver Model: Ein Treiberstandard von Microsoft, um die Entwicklung von Treibern zu ermöglichen, die auf unterschiedlichen Windows-Versionen eingesetzt werden können.

76 Enhanced Audio Streaming Interface

gic von Apple aufgekauft wurde und damit die Windowsplattform nicht mehr unterstützt, wird dem EASI-Standard keine große Zukunft mehr vorausgesagt.

GSIF

Eine weitere Treiberschnittstelle, die hier erwähnt sein soll, ist die GSIF[77]-Schnittstelle, die vom Hersteller *Nemesys/Tascam* speziell für die hauseigene Anwendung *GigaSampler/GigaStudio* entwickelt wurde und auch nur dort Verwendung findet. GSIF minimiert ebenso wie ASIO und EASI die Latenz, um das Echtzeitgefühl beim Arbeiten mit der Giga-Software zu gewährleisten.

CoreAudio

Im Rahmen des 2001 veröffentlichten Apple-Betriebssystems Mac OS X wurde unter dem Namen *CoreAudio* eine komplett neue Audio- und MIDI-Treiberarchitektur[78] eingeführt, nicht zuletzt, um dem Anspruch von Apple gerecht zu werden, Produkte für den professionellen Audiobereich zu entwickeln.

> „In creating this new architecture on Mac OS X, Apple's objective in the audio space has been twofold. The primary goal is to deliver a high-quality, superior audio experience for Macintosh users. The second objective reflects a shift in emphasis from developers having to establish their own audio and MIDI protocols in their applications to Apple moving ahead to assume responsibility for these services on the Macintosh platform."[79]

Dieser Ansatz von Apple zielt eindeutig mehr auf die Unterstützung professioneller Studioanwendungen und weniger auf die Unterstützung von Spieleentwicklern wie bei Microsofts Direct Sound. Betrachtet man die Ziele, die Apple mit der Entwicklung von CoreAudio verfolgte, finden sich dort Begriffe wie „Multi-channel Audio I/O" oder „Application determined latency", die dies deutlich machen. Die Praxis zeigt, dass diese Ziele auch eingehalten werden konnten.

ALSA

ALSA[80] steht für *Advanced Linux Sound Architecture* und ist eine Neuentwicklung der Open-Source-Gemeinde, nachdem viele Linux-Programmierer mit dem alten Audiotreiber-System unter Linux, *Open Sound System*[81] (OSS), unzufrieden waren.

Anders als OSS erlaubt ALSA neben der Unterstützung gängiger Consumer-Soundkarten auch den vollen Zugriff auf die Besonderheiten professioneller Audiokarten. Ziel von ALSA ist außerdem eine Minimierung der Latenz.

Seit der Kernel-Version 2.6 ist ALSA die Standard-Schnittstelle für Sound auf Linux-Systemen. Im Gegensatz zu den propietären Treiberschnittstellen auf anderen Systemen profitiert ALSA von der hohen Aktivität der Open-Source-Gemeinde, und es ist ständig mit neuen Versionen und Verbesserungen zu rechnen.[82]

---

77 Giga Sampler Interface

78 Bis Mac OS 9 gab es mit SoundManager bereits eine ordentliche Treiberschnittstelle, die aber noch einige Wünsche offen lies.

79 Aus [COREAUDIO].

80 http://www.alsa-project.org

81 http://www.opensound.com; OSS existiert als *free* sowie als *commercial* Version. Letztere bietet mehr Profifeatures, kann aber nicht mit ALSA mithalten.

82 Zum Vergleich: ASIO 2.0 wurde seit 1999 nicht mehr weiterentwickelt.

Zusammenfassung

Momentan ergibt sich eine Situation, in der Linux- und Mac-Entwickler mit ALSA bzw. CoreAudio auf Treiberarchitekturen zurückgreifen können, die auf die Anforderungen aus dem Studiobereich reagieren und die von Soft- und Hardwareherstellern große Unterstützung erfahren.

Auf Windows ist nach wie vor ASIO vorherrschend, auch wenn ASIO-Treiber meist nur für professionelle Soundkarten angeboten werden. ASIO4All bietet hier eine Möglichkeit, um auch das Potential weniger professioneller Karten für Profi-Zwecke auszureizen.

Auch für Mac-Software wird ASIO nach wie vor eingesetzt werden, da es den großen Vorteil der Plattformunabhängigkeit bietet, was die Portabilität der Software erhöht. Auch arbeiten noch viele Studios mit Mac OS 9.x, welches CoreAudio nicht unterstützt.

Tabelle 1: Übersicht Audiotreiberschnittstellen[83]

Name	Betriebssysteme	Mehrkanal-unterstützung	Latenz	Verbreitung
MME	Windows	Nein	Sehr hoch	Standard
Direct Sound	Windows	Nein	Gering	Sehr hoch
ASIO	Windows, Mac	Ja	Sehr gering	Im professionellen Bereich hoch
EASI	Windows, Mac	Ja	Sehr gering	Gering
GSIF	Windows, Mac	Ja	Sehr gering	Gering
SoundManager	Mac OS 8.x - 9.x	Nein	Gering	Standard
CoreAudio	Mac OS X	Ja	Sehr gering	Standard
Open Sound System	Linux	Nein	Hoch (Commercial: Gering)	Standard (Commercial: Gering)
ALSA	Linux	Ja	Sehr gering	Standard (ab Kernel 2.6)

3. Anforderungen an Audiosoftware im Studiobereich

Es gibt mit Sicherheit zahlreiche Meinungen darüber, welche besonderen Anforderungen Audiosoftware im Studiobereich erfüllen muss, und eine allgemeingültige Definition lässt sich dafür nicht finden. Daher soll im Folgenden eine Liste der wichtigsten Anforderungen aufgestellt werden, die keinen Anspruch auf Vollständigkeit erhebt und die persönliche Sicht und Erfahrung des Autors widerspiegelt.

Allgemeingültige Anforderungen an Software wie Stabilität, geringer Ressourcenverbrauch oder intuitive Benutzerführung werden an dieser Stelle nicht extra aufgeführt, da sie für Audiosoftware im Studiobereich nicht mehr gelten, als für jede andere Art von Software.

---

83 Vgl. [MACMILLAN].

3.1. Echtzeitfähigkeit

Ein Studioanwender wird von einer Audioapplikation erwarten, dass sie auf jede seiner Eingaben und Änderungen ohne merkliche Verzögerung reagiert. Entscheidend ist das beispielsweise in MIDI-Systemen, wo Noten über ein externes MIDI-Keyboard eingegeben und sofort hörbar gemacht werden sollen. Treten hier hörbare Latenzen auf, ist ein normales Einspielen nicht möglich. Ebenso gibt es Software-Klangerzeuger, die in Echtzeit auf eintreffende MIDI-Daten mit Klangveränderung reagieren sollen. Auch hier sind schon die kleinsten Latenzen intolerabel.[84]

In den meisten samplingbasierten Anwendungen wird ebenfalls eine hohe Interaktivität gefordert, so dass der Anwender seine am Bildschirm gemachten Änderungen unmittelbar zu hören bekommt.

3.2. Unterstützung gängiger Formate und Standards

Mit der Zeit haben sich im Audiobereich zahlreiche Formate und Standards verschiedener Hersteller entwickelt. Einige von ihnen sind in bestimmten Bereichen unumgänglich, andere nur sehr selten anzutreffen. Auch arbeiten inzwischen nur noch wenige Studios mit nur einem einzigen Betriebssystem und den darauf üblichen Formaten. Häufig erfolgt die Produktion auf Macintosh-Systemen und die anschließende Weiterverwertung des Materials über das Intranet auf anderen Plattformen, oft Windows.

Daher wird von professioneller Audiosoftware gefordert, dass sie möglichst viele Formate und Standards unterstützt und zwar sowohl zum Im- als auch zum Export. Ebenso sollte sie auf das Aufkommen neuer Formate vorbereitet sein (➔ Erweiterbarkeit).

Natürlich muss eine professionelle Audioanwendung sämtliche Facetten, die bestimmte Formate mit sich bringen, unterstützen. Bestes Beispiel hierfür sind die unter 1.2.4.1 erwähnten Daten-Chunks. Sie werden von den meisten einfacheren Audioanwendungen ignoriert und gehen beim Abspeichern verloren. Erlaubt ein Format verschiedene Qualitätsparameter wie Sampling-Frequenz oder Samplegröße, sollte die Anwendung zumindest nach oben hin möglichst viele Werte erlauben.

3.3. Erweiterbarkeit

Da sich im Audiobereich immer noch relativ häufig neue Standards für Treiber, Effektschnittstellen, Dateiformate etc. etablieren, ist es wichtig, dass Audiosoftware für den Studiobetrieb nicht nur die derzeit gängigen Schnittstellen unterstützt sondern auch Schnittstellen für Erweiterungen (Plug-Ins) anbietet.

Die Forderung nach Erweiterbarkeit geht Hand in Hand mit der Forderung nach Konfigurierbarkeit. Einige Softwarehersteller bauen in ihre Produkte so umfangreiche Konfigurationsmöglichkeiten ein, dass allein damit von Benutzern neue PlugIns geschaffen werden können[85].

---

84 Vgl. [HAIN], S. 20ff.

85 Paradebeispiel hierfür ist das *Environment* in Emagics Sequencersoftware *Logic Audio*, mit dem von einfachen Anpassungen bis hin zu komplexen PlugIns zahlreiche Anwendungen auf Basis einer grafischen Oberfläche entwickelt werden können.

3.4. Konfigurierbarkeit

Jede Studioumgebung sieht anders aus und jeder Benutzer hat eigene Präferenzen für die Integration einer Software in diese Umgebung. Es ist also wichtig, dass der Benutzer Zugriff auf möglichst viele Parameter hat. Zu diesen Parametern gehören insbesondere

- Auswahl der zu verwendenden Geräte (z.B. Soundkarte, MIDI-Interface), falls mehrere vorhanden sind
- Einfluss auf verwendete Puffergrößen und andere Hardwareparameter
- Auswahl der Festplattenbereiche für temporäre Dateien oder Streaming
- Qualitäts- und Leistungsparameter verwendeter Algorithmen

3.5. Nichtdestruktives Arbeiten

Professionelle Audiosoftware sollte die verschiedenen Ansätze für Nichtdestruktive Bearbeitung (siehe 1.3.2.3) konsequent umsetzen, um somit eine größtmögliche Spontaneität bei optimaler Ressourcennutzung zu gewährleisten.

3.6. Skalierbarkeit

Um skalierbar zu bleiben, muss professionelle Audiosoftware unabhängig von der anfallenden Datenmenge gehalten werden. In der Regel sind viele Programme auch im Audiobereich darauf ausgelegt, alle Bearbeitungsschritte komplett im Hauptspeicher des Computers durchzuführen. Das ist auch gut so, da die Verarbeitung somit viel schneller erfolgen kann, als wenn die Daten für jeden Zugriff von der Festplatte gelesen und wieder geschrieben werden müssten. Im Studiobereich wird aber oft mit so großen Datenmengen hantiert, dass der Arbeitsspeicher früher oder später für diese Vorgehensweise zu klein wäre und teure Seitenfehler[86] oder Out-of-Memory-Fehler die Folge wären (Abbildung 9 zeigt das Ergebnis des Versuchs, eine 95 MB große WAVE-Datei[87] im Windows Soundrecorder auf einem Rechner mit 512 MB Arbeitsspeicher auf halbe Geschwindigkeit zu verringern). Man denke nur an eine 5.1 Surround-Mischung eines Kinofilms. Die Datenmenge für solch ein Projekt kann sehr schnell einige Gigabyte erreichen, wofür derzeit kein Hauptspeicher ausgelegt ist.

---

[86] Unter Seitenfehlern versteht man die Notwendigkeit zum Nachladen nicht im RAM befindlicher Speicherseiten aus dem virtuellen Speicher.

[87] Das entspricht ca. 9:30 Minuten Audio in CD-Qualität.

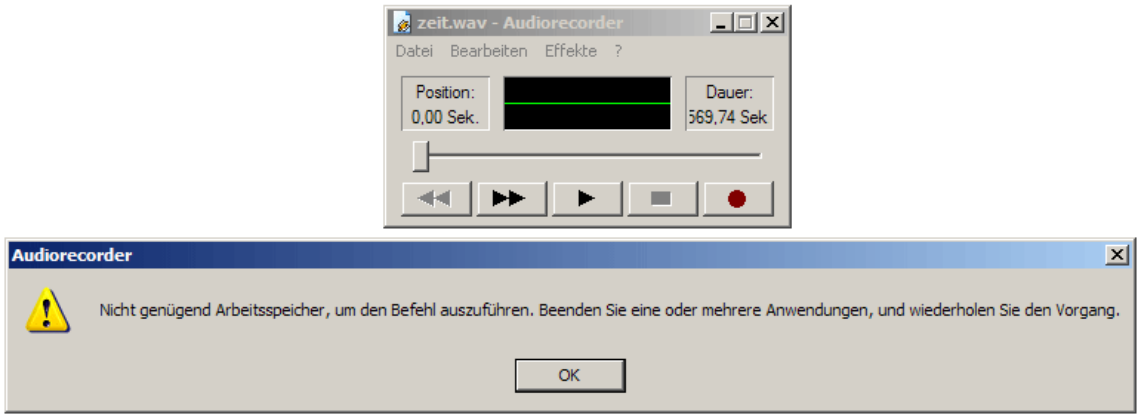

Abbildung 9: Bearbeiten einer 95 MB großen Datei im Windows Soundrecorder

Um mit so großen Datenmengen umgehen zu können, muss die Audiosoftware Streamingmechanismen unterstützen, um jederzeit nur die aktuell benötigten Daten im Hauptspeicher zu halten und bei Bedarf weitere Daten von Festplatte nachzuladen, ohne dabei allerdings die Performance durch die Festplattenzugriffe wesentlich zu beeinträchtigen.

3.7. Nutzen vorhandener Ressourcen

Im Allgemeinen kann man davon ausgehen, dass bei einem Studiocomputer viel Geld in die Hardwareausstattung investiert wurde und dass darüber hinaus spezielle Hardware daran angeschlossen ist, die sich in gängigen Consumer-Rechnern nur selten findet. Zwingende Voraussetzung für professionelle Audiosoftware ist daher, dass die vorhandene Hardwareausstattung voll ausgenutzt werden kann und dass Schnittstellen bestehen, über die auf spezielle Hardware zugegriffen werden kann. Ebenso sollte dem Benutzer die Möglichkeit offen stehen, über die Nutzung seiner Ressourcen soweit wie möglich selbst zu entscheiden (➔ Konfigurierbarkeit).

Von einem einfachen Audioplayer wird niemand erwarten, dass man auswählen kann, welche Kanäle der Soundkarte genutzt werden, da gängige Soundkarten nur über einen Stereoausgang verfügen. Für Software in Studioumgebungen ist hingegen die Unterstützung eventuell vorhandener Mehrkanalhardware zwingend erforderlich.

3.8. Synchronisation

Eine alltägliche Situation in Studios stellt die Synchronisation mehrerer Mediendatenströme zueinander dar. Bestes Beispiel hierfür ist die Vertonung eines Videos, bei der jeder Klang des Audiosignals meist genau einem Ereignis im Bild entspricht, so dass eine genaue Synchronisation erforderlich ist. Hierfür gibt es verschiedene Synchronisationsformate wie beispielsweise SMPTE Timecode, MIDI Time Code oder MIDI Clock.

Wichtig ist auch das Bestimmen der Synchronisationsquelle. Sobald mehrere Hardwaregeräte im Einsatz sind, ergibt sich die Problematik, dass unterschiedliche Clocks im Spiel sind, an denen sich die Geräte orientieren. Es nutzt wenig, wenn in der Software genau angegeben werden kann, zu welchem Videoframe ein bestimmter Klang ertönen und wann welcher MIDI-Befehl gesendet wer-

den soll, wenn Video-, Audio- und MIDI-Hardware unterschiedliche Clocks verwenden (vgl. 1.3.1.2).

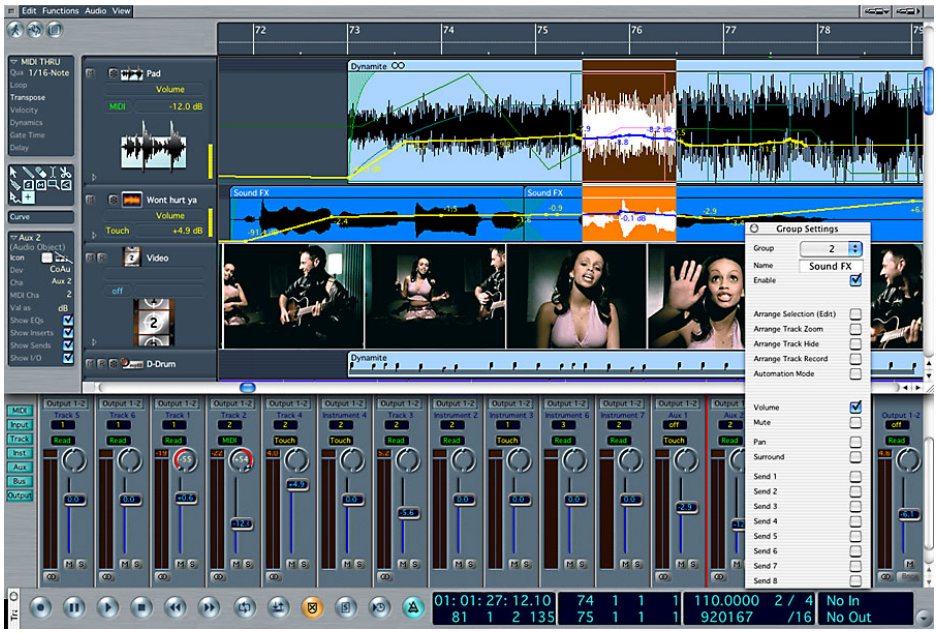

Abbildung 10: Synchronisation mehrerer Mediendatenströme im Audio-/MIDI-Sequenzer Logic Audio[88]

Professionelle Audiosoftware muss daher die Möglichkeit bieten, die Master-Slave-Beziehungen für die Synchronisation festzulegen und darüber hinaus an externe Geräte unterschiedliche Synchronisationsformate zu senden bzw. diese von selbigen zu empfangen.

Ein weiterer Faktor, der für die Erreichung einer perfekten Synchronisation nötig ist, sind die Latenzen der beteiligten Komponenten. Sind diese bekannt, können sie mit einberechnet werden und werden so nicht zu Störfaktoren.

88 Quelle: http://www.emagic.de/

IV. Audioverarbeitung in Java

Im Folgenden werden die Bestandteile und Eigenschaften von Java vorgestellt und besprochen, die für den Bereich Audioprogrammierung relevant sind. Dies sind insbesondere die verschiedenen APIs, die in Java für die Verarbeitung von Audiodaten existieren. Auch wird diskutiert, inwieweit Java für die Erfüllung der im letzten Abschnitt aufgeführten Echtzeitanforderungen förderlich ist oder ein Hindernis darstellt.

1. Audio APIs in Java

In Java gibt es verschiedene Möglichkeiten, auf Audio-Hardware zuzugreifen bzw. Audiodaten zu verarbeiten. Die wichtigsten sollen im Folgenden kurz vorgestellt werden.

Java Speech

Das Java Speech API[89] ist Bestandteil des JDK und wurde darauf ausgelegt, Sprachverarbeitung in Java zu ermöglichen. Sprache kann damit sowohl analysiert als auch synthetisch erzeugt werden. Sun arbeitet in diesem Bereich mit verschiedenen Firmen zusammen, um unterschiedliche Implementierungen zu ermöglichen. Für den Studiobereich spielt diese Art der Sprachverarbeitung allerdings keine Rolle, daher wird auf Java Speech hier nicht näher eingegangen.

Java3D

Java3D[90] ist ein optional erhältliches Paket, das sich – wie der Name schon sagt – der 3D-Programmierung widmet. Mit Java3D ist es möglich, Szenegraphen und andere interaktive dreidimensionale Modelle zu erstellen und zu verwalten. Darüber hinaus bietet es auch Ansätze für 3D-Audio, also die Option, Klangquellen im 3D-Raum zu positionieren.[91] Diese Funktionalität könnte Java3D für Surround- oder Mastering-Anwendungen[92] interessant machen, allerdings erscheint allein dieser beiden Punkt als etwas mageres Argument für die Verwendung von Java3D zur Programmierung professioneller Audioanwendungen, zumal eine Surround-Unterstützung für Spiele ganz anders aussehen kann als beispielsweise für die Vertonung eines Kinofilms.

Java Media Framework

Das Java Media Framework (JMF)[93] ist ein von Sun als Zusatzmodul zum JDK angebotene Multimediaframework, das einen plattformunabhängigen und stark abstrahierten Zugriff auf Medieninhalte ermöglichen soll. Es unterstützt das Abspielen, Aufnehmen, Verarbeiten, Streamen und Speichern zeitabhängiger Daten wie Audio- und Videoströme. Das JMF liegt sowohl in einer reinen Java-Version vor, die auf allen Plattformen eingesetzt werden kann, als auch als optimierte Implementierung für unterschiedliche Plattformen, darunter Windows, Linux und Max OS X.

---

89 http://java.sun.com/products/java-media/speech/

90 http://java.sun.com/products/java-media/3D/

91 Ähnliche Ansätze werden vom OpenGL-Pendant OpenAL (http://www.openal.org/) bzw. dem dazugehörigen Java-Binding JOAL (https://joal.dev.java.net/) verfolgt.

92 3D-Klangeffekte werden auch beim Stereo-Mastering eingesetzt, um den Klangeindruck zu verbessern.

93 http://java.sun.com/products/java-media/JMF/

Als kostenloses, plattformunabhängiges, erweiterbares und objektorientiertes Multimediaverarbeitungsframework ist das JMF derzeit eine einzigartige Entwicklung. Das Framework, das vom Konzept her am ehesten an das JMF heranreicht, ist QuickTime von Apple, für das es mit dem QuickTime for Java API auch eine Java-Anbindung gibt. QuickTime übertrifft das JMF in punkto Funktionsumfang und ist weiter verbreitet. Es ist allerdings nur für Mac und Windows implementiert und erfordert darüber hinaus eine Lizenzierung für jeden Client.[94]

JMF lehnt sich sehr an das Konzept von Java Sound (s. u.) an. Viele Komponenten aus Java Sound finden sich dort wieder, allerdings oft in verallgemeinerter Form, um unabhängig vom Medientyp zu bleiben. Weiterhin wird im JMF das Konzept von Verarbeitungsketten (siehe III.1.3.2.2) konsequent umgesetzt. Java Sound bietet dagegen keine explizite Unterstützung für Verarbeitungsketten.

Mobile Media API

Beim Mobile Media API (MMAPI) handelt es sich um eine reduzierte Version des JMF, die für die Verwendung auf mobilen Endgeräten ausgelegt ist und auf der Java 2 Micro Edition (J2ME) basiert. Das MMAPI gewinnt derzeit stark an Bedeutung für die Entwicklung Java-basierter Multimediaanwendungen für mobile Endgeräte. Es ist als plattformunabhängiges Medienverarbeitungsframework für diese Geräte derzeit konkurrenzlos.[95]

Java Sound

Alle bisher vorgestellten APIs sind eher im Highlevel-Bereich anzusiedeln, da sie zum Ziel haben, die Komplexität der unteren Schicht in der Kommunikation mit der Audio-Hardware vor dem Programmierer zu verbergen und ihm bestimmte wiederkehrende Aufgaben abzunehmen. Java Sound[96] dagegen ist ein Framework, das sich im Lowlevel-Bereich eben dieser Komplexität widmet, indem es dem Programmierer Zugriff auf die Abläufe in dieser Schicht bietet. Java Sound ist seit der Version 1.3 fester Bestandteil des JDK und die meisten anderen APIs verwenden intern Java Sound. In Punkt V wird Java Sound als Hauptinhalt dieser Arbeit detaillierter beschrieben.

2. Echtzeit und Performance in Java

„Java ist langsam." - eine oft geäußerte Kritik, die generell gegen die Nutzung von Java in zeitkritischen Systemen – wie z.B. im Medienumfeld – vorgebracht wird. Es ist zwar nicht von der Hand zu weisen, dass Java durch das Konzept der Virtual Machine einen gewissen Overhead erzeugt und somit einen Nachteil gegenüber Maschinencode zu kompensieren hat, trotzdem sollten solche Aussagen kritisch betrachtet werden.

Zunächst muss zwischen Performance und Echtzeit unterschieden werden. Während Performance die durchschnittliche Ausführungszeit meint, ist mit Echtzeit die Fähigkeit zum Erfüllen harter Echtzeitanforderungen – also das Kontrollieren der maximalen Ausführungszeiten – gemeint.

Ein Grund, warum mit Java nicht immer 100% der vom System her möglichen Geschwindigkeit erreicht werden kann, ist die angestrebte Plattformunabhängigkeit. So muss an manchen Stellen

---

94 Vgl. [EIDENBERGER], S. 8ff.
95 Vgl. [EIDENBERGER], S. 8f.
96 http://java.sun.com/products/java-media/sound/

eine reine Java-Lösung für bestimmte Aufgaben gewählt werden, für die es auf manchen Plattformen performantere native[97] Funktionen gäbe. Bestes Beispiel ist das Package `java.lang.Math`: Die hier verwendeten Fließkommazahlenberechnungen könnten vom Betriebssystem sehr schnell ausgeführt werden. Da das Ergebnis aber nicht auf allen Plattformen Bit für Bit identisch ist, musste die langsamere Variante einer reinen Java-Lösung gewählt werden.[98]

Selbst wenn Java jeden Methodenaufruf per JNI[99] (siehe 3) direkt als Aufruf an das darunter liegende Betriebssystem weitergeben könnte, wäre das mit Performanceeinbußen verbunden, da auch das JNI nicht ganz ohne Overhead auskommt.[100]

Allerdings hat Java in bestimmten Fällen durchaus auch Performancevorteile gegenüber nativen Anwendungen, da beispielsweise Pointer in C/C++ manche Code-Optimierung durch den Compiler verhindern, die in Java möglich wäre. Abbildung 11 zeigt das Ergebnis von Benchmarks der IBM Virtual Machine (IBM), der HotSpot Virtual Machine von Sun (HotSpot) sowie unterschiedlicher C-Compiler bei der Berechnung einer FFT mit unterschiedlichen Array-Größen. *„So what do you conclude from this benchmark? Java is twice as fast as C, or twice as slow, or anything in-between!"*[101] Eine ausführlichere Abhandlung dieser Thematik findet sich bei [LEWIS] und [COWELL-SHAH].

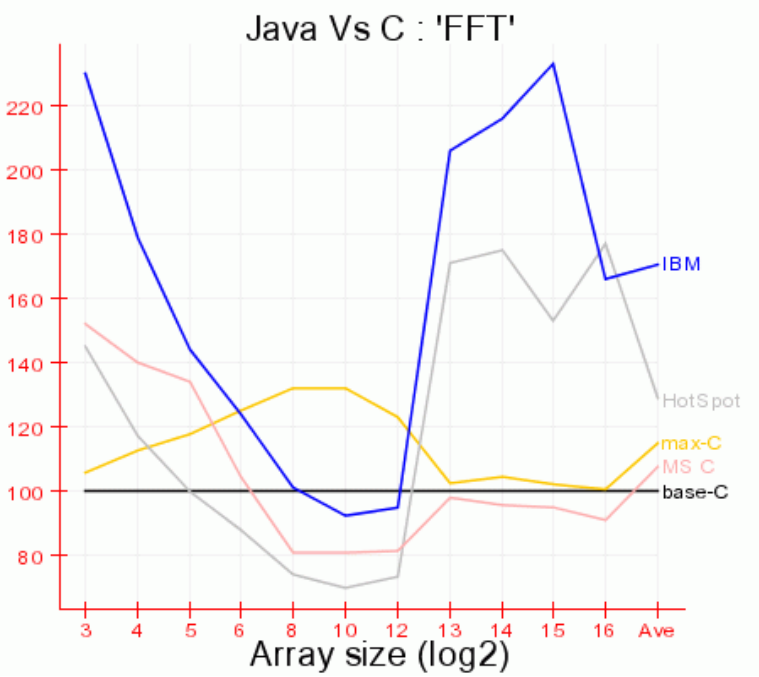

Abbildung 11: Performance-Benchmark unterschiedlicher JVMs und C-Compiler[102]

Auch zeigen die Erfahrungen, dass der größte Performanceverlust vom Programmierer selbst verschuldet ist und nicht von der verwendeten Programmierplattform. So kann beispielsweise ein C-

97 Mit nativ sind in dieser Arbeit solche Softwarebestandteile gemeint, die in systemnäheren Sprachen als Java implementiert sind (vgl. 3).
98 Vgl. [SHIRAZI], S. 67.
99 Java Native Interface
100 Vgl. [SHIRAZI], S. 90.
101 Aus [LEWIS].
102 Quelle: [LEWIS].

Programmierer durch ineffiziente Speicherverwaltung ebenso langsamen Code erzeugen wie ein Java-Programmierer, der laufend neue Objekte erzeugt.

Bei interaktiven Programmen wie Audioapplikationen kommt ein weiterer Aspekt hinzu: *„Empfundene Performance ist letztlich, was zählt."*[103] Gerade in diesem Punkt werden oft Programmiersünden begangen, die weit schwerer wiegen als die Performanceunterschiede zwischen Java und nativem Code. So ist es enorm wichtig, den Benutzer ständig über den aktuellen Fortschritt einer Aktion auf dem Laufenden zu halten (z.B. Fortschrittsbalken, Auslastungsanzeige, etc.) und ihm stets die Möglichkeit offen zu halten, darauf auch aktiv Einfluss zu nehmen (z.B. Wahl der Qualität, Abbruch langer Aktionen, etc.).

Entscheidend für die Performance einer Java-Anwendung kann die Art der Bytecode-Ausführung sein. Ebenso spielt die Garbage Collection eine große Rolle, insbesondere auch für das Echtzeitverhalten. Daher werden im Folgenden zunächst die unterschiedlichen Möglichkeiten zur Bytecodeausführung und anschließend unterschiedliche Garbage Collection Algorithmen besprochen.

2.1. Bytecode-Ausführung

Für die Art des Kompilierens und Ausführens von Java-Bytecode existieren verschiedene Varianten, die zum Teil erheblichen Einfluss auf die Performance haben. Dafür soll noch einmal kurz das Prinzip des Java-Bytecodes vor Augen geführt werden (vgl. Abbildung 12):

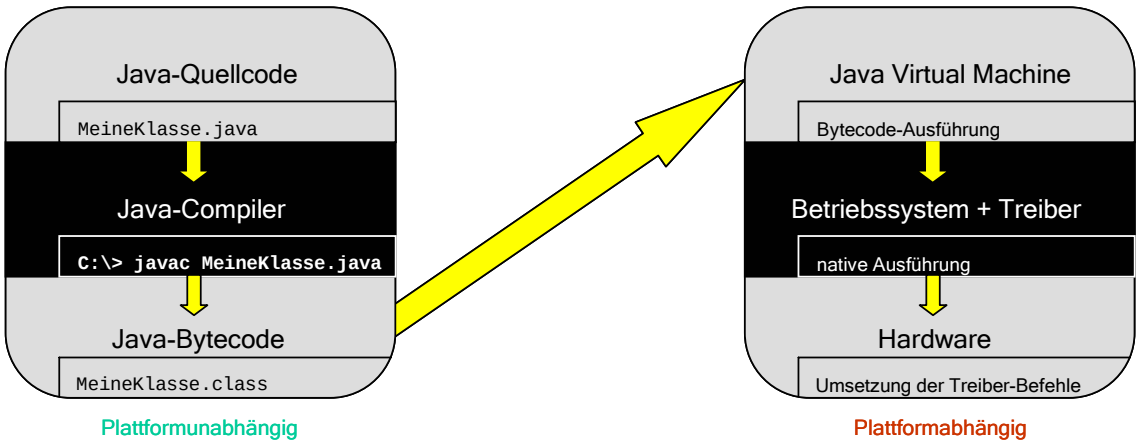

Abbildung 12: Prinzip von Java-Bytecode

Beim Erzeugen der `.class`-Dateien aus Java-Quellcode wird vom `javac`-Compiler ein plattformunabhängiger Bytecode erzeugt, der von jeder Java Virtual Machine (JVM) gelesen werden kann. Zur Ausführung auf dem Zielsystem muss die dortige JVM diesen Bytecode dann so übersetzen, dass er auf dem Zielsystem ausgeführt werden kann. Für diese Übersetzung existieren die folgenden Möglichkeiten:

Interpretieren

Zunächst besteht die Möglichkeit, den Java-Bytecode zur Laufzeit zu interpretieren, was bei der ersten Generation von JVMs noch die Regel war. Bei dieser Variante wird jede Java-Instruktion zur

[103] Aus [SCHREIBER], S. 18.

Laufzeit in eine Reihe nativer Instruktionen übersetzt und ausgeführt. Sie ist allerdings die langsamste Lösung und wird heute höchstens noch dort eingesetzt, wo nur sehr begrenzter Speicher zur Verfügung steht.[104]

JIT

Um die Ausführung zu beschleunigen, wurden JIT[105]-Compiler entwickelt. Diese übersetzen Codeblöcke unmittelbar vor der ersten Ausführung in einen plattformspezifischen Binärcode, der später bei jedem Zugriff auf diesen Codeblock verwendet wird, was die Ausführung entschieden beschleunigt. Allerdings bleibt zum Kompilieren dieses Binärcodes nicht viel Zeit, weshalb auf aufwändige Analysen zur Codeoptimierung verzichtet werden muss. Weiterhin benötigen JVMs mit JIT-Compiler mehr Speicher und eine längere Startzeit als JVMs mit Interpretern, da sie den Binärcode nur im RAM[106] halten. Zudem kann es zu größeren Pausen in der Programmausführung kommen, wenn ein bestimmter Codeblock zum ersten Mal ausgeführt werden soll und zunächst kompiliert werden muss.[107]

DAC

Eine weitere Verbesserung des JIT-Konzepts ist die Dynamic Adaptive Compilation (DAC), die sich in Suns aktuell ausgelieferten JVMs, den so genannten HotSpot-JVMs findet. *„A dynamic adaptive compiler is basically a just-in-time compiler with added intelligence."*[108] Der Name HotSpot kommt vom 80-20-Prinzip, auf dem DAC basiert:

> „Dieses besagt, dass während 80% der Laufzeit eines Programms gewöhnlich 20% des Codes ausgeführt werden. Daraus folgt, dass es sich lohnt, genau jene 20% besonders schnell auszuführen, während Optimierungen in den restlichen 80% des Codes keinen großen Effekt haben. [...] Im Englischen heißen die besonders kritischen Programmteile auch Hotspots – daher der Name."[109]

DAC-JVMs verhalten sich bei der ersten Ausführung eines Codeblocks innerhalb einer JVM-Instanz wie interpretierende JVMs mit der Ausnahme, dass sie die Laufzeiten für die Ausführung analysieren und die Häufigkeit der Aufrufe einzelner Codeblöcke protokollieren. Darauf aufbauend entscheiden sie selbst, welche Teile des Bytecodes zu Binärcode kompiliert und dabei wie stark optimiert werden sollen. Die Ergebnisse der Laufzeitanalysen fließen dabei in die Optimierung ein.[110] Man kann sich leicht vorstellen, dass die Performance mit zunehmender Laufzeit des Programms zunimmt. So ist der Eindruck zu erklären, dass manche Java-Anwendungen kurz nach Programmstart noch etwas träger reagieren, als im weiteren Verlauf der Ausführung. Daraus resultiert aber auch, dass die Algorithmen und Entscheidungen bei der Optimierung für Programme mit kurzen Laufzeiten im Optimalfall anders ausfallen sollten, als für Programme mit langen Laufzeiten. Aus diesem Grund bietet Sun jeweils eine Client HotSpot JVM (für kürzere Laufzeiten) und eine Server HotSpot JVM für lang laufende Serverprozesse an.

---

104 Vgl. [SCHREIBER], S. 33.
105 Just-In-Time
106 Random Access Memory
107 Vgl. [DICKMAN], S. 3f.
108 Aus [DICKMAN], S. 4.
109 Aus [SCHREIBER], S. 33.
110 Für diese Optimierungen stehen unterschiedliche Verfahren wie z.B. *loop unrolling* zur Verfügung, auf die hier nicht näher eingegangen wird.

WAT/AOT

Ein anderer Ansatz als die bisher vorgestellten ist das Übersetzen des gesamten Java-Bytecodes in Binärcode *vor* der Ausführung. Man unterscheidet hier Ahead of Time Compilation (AOT) und Way Ahead of Time Compilation (WAT). AOT wird unmittelbar vor dem Starten des Programms auf der Zielplattform ausgeführt. Das Ergebnis kann für weitere Programmstarts zwischengespeichert werden. WAT findet schon beim Erstellen des Bytecodes statt. Hierbei geht natürlich die Plattformunabhängigkeit verloren, weshalb für jede gewünschte Zielplattform ein WAT-Compile-Vorgang nötig wird.

Für das Erzeugen des Binärcodes ist bei AOT- und WAT-Verfahren quasi keine Zeitbeschränkung gegeben; so können komplexe Analyse- und Optimierungsschritte mit eingebunden werden, was die Qualität des resultierenden Codes zwangsläufig verbessert. AOT findet sich hauptsächlich in eingebetteten Systemen; für WAT gibt es einige Ansätze wie beispielsweise den GNU[111] Java Compiler[112] (GJC), die aber noch nicht für alle Einsatzzwecke uneingeschränkt geeignet sind.

2.2. Garbage Collection

Ein Bestandteil, der Java entschieden von nativen Sprachen wie C/C++ unterscheidet, ist die Art der Speicherverwaltung. Der Programmierer muss sich nicht explizit um das Allozieren und Deallozieren von Speicher kümmern – die JVM übernimmt das komplette Speichermanagement. Das ist zwar zunächst ein großer Vorteil, da gerade dieser Bereich sonst sehr komplex und fehleranfällig ist. Gerade in punkto Echtzeit ist dieses Verhalten aber manchmal auch ein entscheidender Nachteil, da dem Programmierer die Möglichkeit entzogen wird, auf diese performance- und echtzeitkritischen Vorgänge gezielt Einfluss zu nehmen. Hier ist insbesondere das Aufräumen nicht mehr benötigter Objekte gemeint, was in Java der Garbage Collector (GC) erledigt.

Über sein Verhalten können kaum Vorhersagen getroffen werden. Es kann allerdings hilfreich sein, die verschiedenen Garbage Collection Algorithmen zu kennen, da sich einige ihrer Parameter oft über die JVM-Parameter steuern lassen.

Für die meisten GC-Algorithmen ist es nötig, dass der Programmfluss zwischenzeitlich komplett unterbrochen wird, damit der Garbage Collector exklusiven Speicherzugriff erhält und keine Inkonsistenzen – beispielsweise durch Referenzen auf verschobene Objekte – entstehen. *„Dies führt zu unangenehmen Pausen, in denen die Applikation nicht auf Eingaben reagiert. Diese Pausen sind für Echtzeitanwendungen (z.B. Audio-/Video-Anwendungen [...]) nicht akzeptabel."*[113] Daher gilt es, für Audioanwendungen – falls möglich – einen GC-Algorithmus auszuwählen, der diese Pausenzeiten gering hält.

2.2.1 GC-Algorithmen

Kopierender Kollektor

Die einfachste Implementierung eines Garbage Collectors, die in Java Verwendung findet, ist der Kopierende Kollektor. Hierbei wird der Speicher in zwei oder mehr Teile aufgeteilt, wovon zu jeder Zeit nur ein Teil gültige Objekte enthält. Bei jedem GC-Durchlauf wird der Objektbaum von der

111 GNU´s Not UNIX
112 http://gcc.gnu.org/java/
113 Aus [SCHREIBER], S. 43.

Wurzel an durchlaufen und alle erreichbaren Objekte werden in den nächsten Speicherteil kopiert, der anschließend zum aktuell gültigen Teil wird. Der gerade durchlaufene Speicherteil kann nach dem Kopieren komplett gelöscht werden. Kopierende Kollektoren führen zu schnellen Allokationszeiten, da der freie Speicher nicht fragmentiert wird und daher keine aufwändige Verwaltung des freien Speichers nötig ist. Allerdings benötigen sie insgesamt relativ viel Speicher und können zu langen Pausenzeiten während der Garbage Collection führen.[114]

Mark & Sweep / Mark & Compact

Bei diesen beiden Verfahren wird bei jedem GC-Durchlauf der gesamte Objektbaum durchlaufen und jedes erreichbare Objekt mit einem Bit markiert (sog. *Mark-Phase*). Anschließend wird in der sog. *Sweep-Phase* der gesamte Speicher durchlaufen, und alle nicht markierten Objekte werden gelöscht. Hierdurch wird allerdings der Speicher fragmentiert, was sich negativ auf die Allokationszeiten neuer Objekte auswirkt.

An dieser Stelle unterscheidet sich der Mark & Compact Algorithmus von Mark & Sweep: Er führt statt der Sweep-Phase eine *Compact-Phase* durch, in der nicht markierte Objekte gelöscht und gleichzeitig die lebendigen Objekte so im Speicher verschoben werden, dass keine Fragmentierung entsteht. Daher benötigt Mark & Compact etwas länger für die Speicherbereinigung als Mark & Sweep, führt aber anschließend zu besserem Laufzeitverhalten.[115]

2.2.2 Optimierung der Garbage Collection

Sämtliche im vorigen Abschnitt vorgestellten Algorithmen erfordern einen exklusiven Zugriff auf den gesamten Speicher der Anwendung und müssen diese daher für die Zeit der Speicherbereinigung komplett anhalten, wodurch die erwähnten unerwünschten langen Pausen entstehen können. Um dies zu vermeiden, werden die Algorithmen in ihrer Verwendung optimiert.

Generationen-Kollektoren

In der Programmier-Realität ist es keineswegs so, dass alle Objekte eine in etwa gleiche Lebensdauer haben und daher von der Speicherbereinigung gleich betroffen sein können. Vielmehr hat sich gezeigt, dass die meisten Objekte eine sehr kurze Lebensdauer haben, wohingegen einige wenige sehr lange existieren. Diese Erkenntnis führte zur Entwicklung von Generationen-Kollektoren. Sie unterteilen den verfügbaren Speicher in Bereiche unterschiedlicher Größen, die so genannten Generationen.

Neue Objekte werden stets in der „jüngsten“ Generation erzeugt und falls sie mehrere Bereinigungsschritte überleben in eine „ältere“ Generation verschoben. Sobald der Speicher einer Generation an die Grenzen seiner Kapazität kommt, wird er mit einem der oben beschriebenen Algorithmen aufgeräumt.

Der Vorteil von Generationen-Kollektoren ist, dass nicht bei jedem Bereinigungsvorgang der gesamte Speicherbereich gesäubert werden muss, was die Pausenzeiten drastisch verringern kann, zumal wenn die Größe der Generationen geschickt gewählt wird.[116]

---

114 Vgl. [SCHREIBER], S. 41f.
115 Vgl. [SCHNEIDER], S. 24ff.
116 Vgl. [SCHREIBER], S. 43.

Inkrementelle und nebenläufige Speicherbereinigung

Bei der inkrementellen Speicherbereinigung werden die Pausenzeiten kurz gehalten, indem darauf verzichtet wird, bei jedem Durchlauf den gesamten Speicher bereinigen zu wollen; vielmehr beschränkt man sich jeweils auf die Bereinigung kleiner Teilbereiche. Einen ähnlichen Ansatz versucht die nebenläufige Speicherbereinigung: Hierbei läuft die Garbage Collection in einem eigenen Thread parallel zur eigentlichen Programmausführung.

Da sich der Zustand der nicht untersuchten Speicherteile ständig ändern kann – und in der Regel auch häufig wird –, muss die Garbage Collection über diese Änderungen informiert werden, was einen hohen Synchronisationsaufwand erfordert. Ergebnisse von inkrementeller und nebenläufiger Speicherbereinigung sind daher geringere Pausenzeiten, aber schlechtere Performance zur Laufzeit.[117]

> „Falls Ihr Programm Echtzeit-Kriterien standhalten muss, wählen Sie eine VM mit inkrementellem oder nebenläufigem Garbage Collector. Inkrementelle Garbage Collection hält die maximalen Pausenzeiten kurz, führt jedoch sonst meist zu schlechterer Performance."[118]

2.2.3 Beeinflussung der Garbage Collection

Fast jede JVM-Implementierung bietet die Möglichkeit, über Parameter das Verhalten des Garbage Collectors zu beeinflussen. Suns aktuelle HotSpot-JVMs basieren beispielsweise auf einem Generationen-Kollektor, der den Heap in fünf Bereiche aufteilt und standardmäßig einen kopierenden Kollektor für die jüngste Generation sowie einen Mark & Compact Algorithmus für die älteren Generationen verwendet. Über JVM-Parameter kann hier u.a. das Größenverhältnis der Generationen zueinander bestimmt und der reine Mark & Compact Algorithmus bei Bedarf durch inkrementelle Speicherbereinigung ersetzt werden. Ebenso besteht die Möglichkeit, sich die Aktivität des Garbage Collectors ausgeben zu lassen, um die optimale Einstellung für eine Applikation zu finden.[119]

Darüber hinaus bietet Java die Methode `System.gc()` an, die der JVM mitteilt, wann ein günstiger Zeitpunkt für eine Speicherbereinigung gekommen ist. Allerdings ist keineswegs gewährleistet, dass die Speicherbereinigung tatsächlich mit Aufruf dieser Methode ausgeführt wird; sie ist vielmehr als Hinweis zu verstehen. Außerdem sollte der Aufruf mit Vorsicht eingesetzt werden, da er eine vollständige Speicherbereinigung – und damit eine lange Programmunterbrechung – nach sich ziehen kann, auch wenn sie nicht unbedingt nötig wäre.[120]

2.3. Konzepte zum Performancegewinn

Der Java-Programmierer ist nicht hilflos der Virtual Machine und ihren Eigenheiten ausgeliefert. Dieser Abschnitt zeigt einige Konzepte, die beim Java-Programmieren aus Performancesicht beachtet werden sollten, um beispielsweise die Aktivität des Garbage Collectors gering zu halten.

---

117 Vgl. [SCHREIBER], S. 43.

118 Aus [SCHREIBER], S. 49.

119 Für eine komplette Auflistung der möglichen JVM-Parameter zur Beeinflussung des Garbage Collectors bei der HotSpot-JVM siehe [SCHREIBER], S. 45.

120 Vgl. [SCHREIBER], S. 83.

Vermeiden unnötiger Objekterzeugung

Das Erzeugen neuer Objekte ist teuer: Neuer Speicher muss alloziert werden, der Konstruktor muss ausgeführt werden und der Garbage Collector hat später mehr zu tun. Natürlich ist das Erzeugen neuer Objekte eine der Hauptaufgaben in einem Java-Programm und das soll auch so sein; allerdings gibt es Mechanismen, um überflüssige Objekterzeugungen zu vermeiden, z.B.:

- ***StringBuffer***
  Die Klasse `StringBuffer` bietet zahlreiche Methoden für das Zusammenfügen oder Zerteilen von Zeichenketten, die im Gegensatz zu den `String`-Methoden oder dem `+`-Operator nicht für jede Aktion ein neues `String`-Objekt erstellen. Für häufige Manipulationen von Zeichenketten sollte daher stets die `StringBuffer`-Klasse der `String`-Klasse vorgezogen werden.

- ***Collections***
  Sämtliche `Collection`-Klassen bieten die Möglichkeit, bei ihrer Erzeugung die erwartete maximale Größe anzugeben. Ist diese in etwa bekannt, kann dadurch verhindert werden, dass beim Wachsen der `Collection` im Hintergrund laufend neue Objekte für die interne Datenhaltung der `Collection` erzeugt und deren Inhalte kopiert werden müssen.

- ***Ökonomischer Einsatz von Exceptions***
  `Exception`-Objekte zu erzeugen dauert etwas länger als das Erzeugen gewöhnlicher Objekte, da immer der aktuelle Stack-Trace ermittelt werden muss. Außerdem können `Exception`s einige wichtige Codeoptimierungen des Compilers verhindern. Daher kann der häufige Einsatz von `Exception`s – abgesehen davon, dass `Exception`s nur für selten auftretende Ausnahmesituationen eingesetzt werden sollten – zu Performanceeinbußen führen. Es kann sich in diesen Fällen durchaus lohnen, die `Exception`-Objekte nicht jedes Mal neu zu erzeugen, sondern bereits geworfene `Exception`s wieder zu verwenden. Man muss dabei allerdings bedenken, dass der in der `Exception` enthaltene Stack-Trace dann seine Gültigkeit verliert.

  Eine weitere Maßnahme zur Vermeidung unnötiger Neuerzeugungen von **`Exception`**s ist der Einsatz einer sinnvollen Vererbungshierarchie für die Implementierung eigener **`Exception`**-Klassen[121]. So empfiehlt es sich, wo immer es angebracht erscheint, von den im JDK enthaltenen **`Exception`**-Klassen zu erben. Dieses Vorgehen verhindert in vielen Fällen, dass **`Exception`**s abgefangen werden müssen, nur um eine neue **`Exception`** eines anderen Typs zu werfen.[122]

Threads bewusst einsetzen

Auf der einen Seite kann der Einsatz von Threads dazu führen, die Performance einer Applikation – insbesondere die empfundene Performance – zu erhöhen, da sie die quasi-parallele (auf Mehrprozessormaschinen sogar tatsächlich parallele) Ausführung von Programmteilen ermöglichen. Die gut realisierte Thread-Unterstützung ist daher auch eine der Stärken von Java. Durch die höhere Kom-

---

[121] Vor der Implementierung eigener `Exception`-Klassen sollte zunächst geprüft werden, ob nicht bereits im JDK enthaltene `Exception`-Klassen verwendet werden können. Ist dies möglich, sollten diese selbstimplementierten Klassen vorgezogen werden.

[122] Vgl. [SCHREIBER], S. 141ff.

plexität und die in vielen Fällen erforderliche Synchronisation zwischen Threads können sie auf der anderen Seite aber auch zu Performanceeinbußen führen, wenn sie ineffizient eingesetzt werden.

Java bietet die Möglichkeit, Objektzugriffe von mehreren Threads über das Schlüsselwort `synchronized` zu synchronisieren. Dies ist auch ein sinnvoller Mechanismus, da beim unsynchronisierten Arbeiten mit mehreren Threads fast zwangsläufig Inkonsistenzprobleme entstehen. Allerdings verleitet die einfache Verwendung dazu, im Zweifel alles zu synchronisieren, um kein Risiko einzugehen. Die Ausführung von `synchronized`-Blöcken kostet aber Performance, weil bestimmte Optimierungen durch den Compiler darin nicht möglich sind. Zu viel Synchronisation kann also leicht zu Performanceproblemen führen.

Um das zu vermeiden, sollten nur die Programmabschnitte in einen `synchronized`-Block gestellt werden, die tatsächlich Synchronisation erfordern. Meist sind das nicht ganze Methoden, obwohl es für den Programmierer natürlich einfacher ist, einer Methodendeklaration ein `synchronized` hinzuzufügen, als bestimmte Teile in einen `synchronized(myObject){...}`-Block zu stellen.

Ebenso kann es in vielen Fällen sinnvoll sein, die Synchronisation dem Aufrufer zu überlassen. Das ist auch der Grund, warum alle Standard-Implementierungen des `Collection`-Interfaces im Gegensatz zur früher verwendeten `Vector`-Klasse ohne Synchronisation geschrieben sind und diese dem Programmierer überlassen.

Ein anderer Aspekt, der im Zusammenhang mit Threads Beachtung finden sollte, ist, dass die `Thread`-Klasse ursprünglich für eine einmalige Verwendung konzipiert ist: Ein `Thread`-Objekt wird mit `new` erzeugt, mit `start()` gestartet, führt eine `run()`-Methode aus und wird nach deren Beendigung terminiert. Das `Thread`-Objekt hat dann keinen Nutzen mehr, da ein Thread nach Beendigung nicht wieder gestartet werden kann. Das Erzeugen und Starten von Threads kostet aber um ein vielfaches mehr Zeit, als das Erzeugen eines einfachen Objekts. Daher kann es sich durchaus lohnen, die `run()`-Methode von Threads so zu implementieren, dass der Thread nach Beendigung einer Aufgabe wartet, bis ihm eine neue Aufgabe zugewiesen wird, anstatt die `run()`-Methode zu verlassen. Dies spart die Kosten des Erzeugens und Startens und ist insbesondere dann sinnvoll, wenn relativ häufig relativ kurze Aufgaben von einem zusätzlichen Thread ausgeführt werden sollen.[123]

Dokumentation und Quellcode lesen

„Die Wahrheit steht im Code. Und kein ernstzunehmender Entwickler kann es sich leisten, die Wahrheit zu ignorieren.“[124] Oft finden sich in der Dokumentation oder im Quellcode Details, die helfen können, die aus Performancesicht beste Lösung für ein Problem zu finden. Daher ist es wichtig, – soweit verfügbar – stets Zugriff auf den Quellcode und die Dokumentation (Javadoc) der verwendeten Bibliotheken zu haben. Ansonsten wird oft der kürzeste Code als der performanteste angesehen, was durchaus nicht immer zutreffend sein muss.

Java-unabhängige Maßnahmen

Neben den aufgeführten Java-spezifischen Konzepten existieren natürlich noch solche, die für jede andere Programmierplattform ebenso gelten, wie Buffering oder Caching. Werden diese nicht be-

123 Vgl. [SCHREIBER], S. 187ff.
124 Aus [SCHREIBER], S. 21.

wusst eingesetzt, hilft auch die performanteste Ausführungsmethode nichts. Auf diese Mechanismen soll jedoch an dieser Stelle nicht näher eingegangen werden.

2.4. Zusammenfassung

Zusammenfassend kann gesagt werden, dass Java zwar mit ein paar Nachteilen behaftet ins „Performance-Rennen“ geht, insbesondere dort, wo es nicht nur um reine mathematische Berechnungen geht. Jedoch können diese Nachteile durch effiziente Programmierung und – falls möglich – die richtige Wahl von JVM und GC-Parametern so in Grenzen gehalten werden, dass sie in vielen Fällen verkraftbar sind. Schwieriger wird es bei der Erfüllung von harten Echtzeitanforderungen. Insbesondere die Garbage Collection kann hier Probleme verursachen, auch wenn für viele Fälle die Wahl der richtigen JVM und das Anpassen des GC-Algorithmus Abhilfe schaffen kann. Es bleibt daher eine der größten Schwächen von Java, dass die Optimierung von Java-Anwedungen für zeitkritische Systeme einen hohen Aufwand mit sich bringt und nicht immer den gewünschten Effekt erzielt. Dabei sollte berücksichtigt werden, dass meist nur entweder auf Echtzeitfähigkeit – also kurze Pausen – oder auf Performance optimiert werden kann.

2.5. Exkurs: SWT/JFace als Alternative zu AWT/Swing

Ein Bestandteil von Java, der bei Desktop-Anwendungen beim Benutzer häufig den Eindruck von Langsamkeit erzeugt, ist das plattformunabhängig gehaltene GUI[125]-Framework AWT[126]/Swing. Die Idee bei Swing ist, das Erscheinungsbild von Desktop-Applikationen unabhängig von dem darunter liegenden Betriebssystem kontrollieren zu können. Daher läuft das Rendering und die Ereignisverarbeitung zu einem Großteil in Java ab und ist in der Regel etwas langsamer, als die betriebssystemeigenen GUI-Frameworks. Da der Benutzer im Allgemeinen bei GUI-Elementen sehr empfindlich auf Verzögerungen reagiert, wirken AWT-/Swing-basierte Anwendungen oft langsam oder träge.[127] Swing wird auch als ein Heavyweight-Framework bezeichnet, da es zwangsläufig einen hohen Ressourcenverbrauch hat.

Um auch Java Applikationen mit „nativem Feeling“ zu ermöglichen, wurde mit dem von IBM initiierten *eclipse*[128]-Projekts unabhängig von AWT/Swing ein GUI-Framework namens SWT[129] aufgebaut, das als Lightweight-Framework konzipiert ist und sich nur als transparente Schicht zu den Betriebssystemaufrufen versteht. Dazu existiert mit JFace auch eine Klassenbibliothek, die auf SWT aufbaut und dem Programmierer häufig wiederkehrende Aufgaben abnimmt. Bei SWT wird bewusst in Kauf genommen, dass der Programmierer das Aussehen der GUI-Elemente nicht vollständig kontrollieren kann. Dieser Verzicht wird durch eine verzögerungsfreiere und damit angenehmere Bedienbarkeit der Programme belohnt.

> „Das SWT unterscheidet sich [...] [bei der Ansprechgeschwindigkeit] nicht von nativen Applikationen, da direkt auf die Ereignisverarbeitung des jeweiligen Betriebs- oder Fenstersys-

125 Graphical User Interface

126 Abstract Window Toolkit

127 An dieser Stelle soll nicht unerwähnt bleiben, dass Sun bei der Entwicklung von JDK 1.5 in den AWT/Swing-Bereich viel Arbeit investiert hat, um diesen Umstand zu verbessern. Inwieweit dies zu einer besseren Performance führt, wurde im Rahmen dieser Arbeit nicht überprüft.

128 http://www.eclipse.org

129 Standard Widget Toolkit

> tems aufgesetzt wird. Swing ist dagegen in der Interaktion oft etwas träge. Zudem ist SWT nicht so ressourcenhungrig wie Swing.“[130]

SWT-Implementierungen existieren derzeit für Windows, Linux (GTK oder Motif), Mac OS X sowie verschiedene Unix-Derivate. Hierbei sind aufgrund des Konzepts von SWT leichte Variationen im Verhalten auf den unterschiedlichen Plattformen unvermeidlich.[131]

Da in Medienapplikationen eine absolut verzögerungsfreie Reaktion der Applikation auf Benutzerinteraktion angestrebt wird, bietet sich SWT/JFace in diesem Bereich als Alternative zu AWT/Swing an. Die Anwendungen im praktischen Teil dieser Arbeit sind daher auch ausnahmslos mit SWT/JFace realisiert.

3. Das Java Native Interface

Es gibt verschiedene Situationen, in denen Programmierer an die Grenzen von Java stoßen und eine Schnittstelle benötigen, um auf Anwendungsteile zuzugreifen, die in einer anderen Sprache geschrieben werden oder wurden. Insbesondere in folgenden Fällen:

- Die gewünschte Funktionalität besteht bereits als funktionierende und bewährte Bibliothek in einer anderen Sprache und eine Portierung in Java wäre zu aufwändig oder aufgrund von fehlendem Know-how unmöglich.
- Es wird ein Zugriff auf spezielle hardware- oder betriebssystemspezifische Funktionen gewünscht, der direkt aus Java heraus unmöglich ist.
- Für eine zeitkritische Anwendung konnte in einer anderen Sprache eine entschieden höhere Ausführungsgeschwindigkeit als in Java erreicht werden.[132]

Dass vor allem die ersten beiden Fälle bei der Entwicklung professioneller Audioanwendungen in Java häufig auftreten, wird im praktischen Teil dieser Arbeit deutlich. Jedoch sollte vor der Verwendung von nativem Code stets eine Abwägung stattfinden, ob der erwartete Vorteil die Einschränkung der Plattformunabhängigkeit sowie den erzeugten Overhead wert ist.

Zwar kann der native Code theoretisch in jeder anderen Programmiersprache geschrieben sein, solange man auf eine Java-Bindung für den entsprechenden Compiler zurückgreifen kann. In den meisten Fällen wird es sich dabei aber um C oder C++ handeln. Für diese Fälle sieht das JDK das Java Native Interface (JNI) vor.

Methoden können in Java mit dem Schlüsselwort `native` versehen werden, um sie als native Methoden zu markieren. Diese werden dann nicht in Java ausimplementiert sondern wie bei einer Interfacedeklaration mit einem Semikolon abgeschlossen (vgl. Codebeispiel 1).

---

130 Aus [DAUM], S. 124.
131 Vgl. [DAUM], S. 124f.
132 Vgl. [HORSTMANN], S. 958.

```
1   /**
2    * JNI-Beispielklasse
3    */
4   class NativeExample{
5     // Bei der ersten Verwendung der Klasse
6     // native Bibliothek laden
7     static{
8       System.loadLibrary("examplelib");
9     }
10    /**
11     * Diese Methode wird in C/C++ ausimplementiert.
12     */
13    public native int nativeMethod();
14  }
```

Codebeispiel 1: JNI-Beispielklasse in Java

Mit dem im JDK enthaltenen Programm `javah` kann man sich dann eine C-Headerdatei erzeugen lassen, die die Deklaration der in C/C++ zu implementierenden Funktionen[133] enthält. Nachdem die Funktionen implementiert und in eine Bibliothek kompiliert wurden, muss diese noch mittels der statischen Methode `System.loadLibrary(String libname)` in Java geladen werden.

Anschließend kann auf die nativen Methoden so zugegriffen werden, als wären es reine Java-Methoden. Nach außen hin bleibt die Verwendung des JNI also transparent. Vom nativen Code aus kann auf die Java-Klassen und –Objekte mit Hilfe des bei jedem Funktionsaufruf von Java übergebenen `JNIEnv`-Pseudoobjekts zugegriffen werden. Ebenso können neue Java-Objekte instanziiert und `Exception`s geworfen werden. Eine direkte Abbildung der Java-Klassen auf C++-Klassen ist dagegen nicht vorgesehen.

Um auf die verschiedenen Besonderheiten, die beim Arbeiten mit nativem Code auftreten können, einzugehen, bietet das JNI darüber hinaus einige spezielle Funktionen an: So existiert mit dem Invocation API eine Möglichkeit, von C/C++-Code aus eine neue JVM zu instanziieren. Ebenso ist es möglich, einen nicht von Java erzeugten Thread an eine existierende JVM-Instanz zu koppeln, um von ihm aus auch Java-Code ausführen zu können. Hierdurch wird es möglich, auf Callbacks – beispielsweise von Treibern – in Java zu reagieren. Ebenso bietet das JNI die Möglichkeit, in nativem Code neue Instanzen der `java.nio.ByteBuffer`-Klasse zu erstellen, die Zugriff auf einen vorgegebenen Speicherbereich haben. So können Daten effizient und ohne teures Kopieren zwischen Java und anderen Prozessen ausgetauscht werden.

[133] Die Funktionsnamen setzen sich aus dem Präfix `Java_` sowie dem vollqualifizierten Klassennamen und dem Methodennamen zusammen.

V. Java Sound

1. Einführung

1.1. Was ist Java Sound?

Java Sound versteht sich als Lowlevel-Framework, das Java-Programmierern einen für Java-Verhältnisse relativ hardwarenahen Zugriff auf vorhandene Sound-Hardware (Sampling und MIDI) ermöglichen soll. Es dient außerdem als Basis für fast alle Implementierungen von Sound-Zugriffen anderer Java Media APIs (vgl. IV.1). Seit JDK 1.3 (1999) ist es ein fester Bestandteil des JDK, welches davor nur sehr begrenzte Sound-Funktionalität bot.[134]

Java Sound umfasst folgende Bestandteile:

- ***Java Sound API***
  Das Java Sound API umfasst die Programmierschnittstelle, die in den Packages `javax.sound.*` definiert wird. Sie ist unabhängig von jeder Implementierung.
- ***Java-Sound-Implementierungen***
  Neben der Referenzimplementierung von Sun für die Zielplattformen Linux, Windows und Solaris existieren auch andere Implementierungen von Drittanbietern. Die wohl bekannteste ist die Linux-Implementierung des Open-Source-Projekts Tritonus[135], die zumindest bis vor JDK 1.5 einige Schwächen der Sun-Implementierung umgehen konnte.
- ***Java Sound Audio Engine***
  Das JDK beinhaltet mit der Java Sound Audio Engine, die auf der Beatnik Audio Engine basiert, einen virtuellen Player für Audio und MIDI, der bereits eine MIDI-Soundbank mitbringt. In früheren Java Sound Versionen war es nur möglich, über die Java Sound Audio Engine Klänge abzuspielen.

1.2. Zielsetzung und Entwicklung von Java Sound

Zur ursprünglichen Zielsetzung bei der Entwicklung von Java Sound finden sich einige bemerkenswerte Zitate in einem Artikel ([MELOAN]), der zu einer Zeit erschien, als an der ersten Release-Version von Java Sound gearbeitet wurde. Der damalige Java Media Manager Michael Bundschuh sah die Bedeutung von Java Sound damals so:

> "Currently, we are enabling multimedia in the computer desktop market by adding true sound support in the Java 2 platform. In the future, we would like to see our Java Sound API technology used in professional, consumer and Internet audio applications."[136]

---

134 Die `Applet`-Klasse beispielsweise war über die `play()`-Methode in der Lage, Sound-Dateien (nur in Suns `.AU`-Format) abzuspielen.

135 http://www.tritonus.org

136 Aus [MELOAN].

Für künftige Versionen wurde sogar von expliziter Unterstützung professioneller Anforderungen gesprochen:

> "Movie and record companies need professional quality audio to use the Java Sound API technology. Implementing support for 24-bit audio and multi-channel configurations in the Java Sound Engine will encourage the development of professional editing and playback applications."[137]

In erster Linie aber sollte zunächst eine brauchbare Schnittstelle zum Zugriff auf Sound-Hardware geschaffen werden. *„Extensive low level support functions will be available so that programmers can input and output sound, control MIDI instruments, and query system functions."*[138] Dieses Vorhaben stellte sich aber aufgrund der angestrebten Plattformunabhängigkeit als sehr schwierig heraus. Gefunden wurde eine erste Lösung mit der Lizenzierung der Beatnik Audio Engine, die elementare Aufgaben wie Sampling- und MIDI-Wiedergabe betriebssystemunabhängig lösen konnte. Die damit erreichte Plattformunabhängigkeit hatte jedoch ihren Preis: Ein Zugriff auf andere auf einem System installierte Audio- oder MIDI-Treiber war nicht möglich, da die Java Sound Audio Engine nicht umgangen werden konnte.

Nach Veröffentlichung der ersten Java-Sound-Version im JDK 1.3 verließen die Java-Sound-Entwickler Sun nach und nach, woraufhin ein Jahr lang niemand bei Sun für Java Sound zuständig war. Erst durch die Anstellung von Florian Bömers als neuem Zuständigen für Java Sound im Jahr 2001 unmittelbar vor der Veröffentlichung des JDK 1.4 wurde Java Sound wieder zum Leben erweckt. Jedoch konnten naturgemäß nicht mehr viele Veränderungen in diese Version übernommen werden.

So kam es, dass bis Version 1.5 nicht weiter über die Zielsetzung von Java Sound nachgedacht wurde. *„erst jetzt kann Sun von einer Referenz-Implementierung sprechen. Und erst jetzt ist es sinnvoll, über wirklich große Änderungen nachzudenken."*[139]

2. Kurzbeschreibung des API

Entsprechend den unter III.1.2 dargestellten beiden Möglichkeiten, Klänge digital darzustellen, gliedert sich das Java Sound API in die beiden Packages `javax.sound.sampled`, welches den Umgang mit samplingbasiertem Audiomaterial ermöglicht und `javax.sound.midi` für MIDI-Funktionalität. Für beide Packages existiert mit den Klassen `AudioSystem` bzw. `MidiSystem` jeweils eine zentrale Klasse, die das Abfragen installierter Geräte sowie einige andere zentrale Aufgaben durch eine Reihe statischer Methoden anbietet.

Hinzu kommen die Packages `javax.sound.sampled.spi` und `javax.sound.midi.spi`, die Drittanbietern die Erweiterung der Implementierung durch PlugIns, so genannte *Service Provider Interfaces (SPI)* ermöglichen.

Im Folgenden werden kurz die wichtigsten Bestandteile des APIs besprochen. Eine vollständige API-Beschreibung findet sich neben [JAVADOC] in [JAVASOUND].

---

137 Aus [MELOAN].
138 Aus [MELOAN].
139 Aus [INTERVIEW].

2.1. Das Package javax.sound.sampled

Die Line-Interface-Hierarchie

Das Zentrum des `sampled`-Package bildet das `Line`-Interface. *„The Line interface represents a mono or multi-channel audio feed. A line is an element of the digital audio "pipeline," such as a mixer, an input or output port, or a data path into or out of a mixer."*[140] Eine `Line` repräsentiert also ein Objekt, durch das Audiodaten fließen können. Subinterfaces von `Line` sind `Port`, `Mixer` und `DataLine`.

`Port`s stehen für tatsächliche Ein- und Ausgangsanschlüsse der Soundkarte, wie den Mikrophon- oder Line-Out-Anschluss. Über `Port`-Instanzen können diese Anschlüsse ein- und ausgeschaltet oder – falls unterstützt – in Lautstärke, Panorama oder ähnlichem kontrolliert werden (siehe auch `Controls`, weiter unten).

Codebeispiel 2 zeigt, wie man mit Java Sound den Mikrophoneingang einer Soundkarte einschalten kann, was sonst üblicherweise über einen Dialog des Soundkartentreibers geschieht: Die Klasse `Port.Info` enthält einige statische Felder, wie z.B. `Port.Info.MICROPHONE`, um Anschlusstypen zu identifizieren. Bis Zeile 5 wird die `Port`-Instanz ermittelt, die zum Mikrophonanschluss gehört und in Zeile 8 wird dieser Anschluss geöffnet.

```
1   // Mixer instanziieren
2   Mixer mixer = ...
3   if(mixer.isLineSupported(Port.Info.MICROPHONE)){
4       // Port-Instanz vom Mixer holen
5       Port mic = (Port)mixer.getLine(Port.Info.MICROPHONE);
6       try{
7           // Anschluss öffnen
8           mic.open();
9       } catch(LineUnavailableException e){
10          // Fehlermeldung
11      }
12  } else{
13      // Fehlermeldung
14  }
```

Codebeispiel 2: Öffnen des Mikrophonanschlusses einer Soundkarte

Allerdings können `Port`-Instanzen nicht verwendet werden, um Daten über diese Anschlüsse zu senden oder aufzunehmen. Vielmehr tauschen `Mixer` und Anwendung über `DataLine`-Instanzen Audiodaten aus; der `Mixer` stellt dann die Verbindung zur Hardware bzw. zum Treiber her. Um `Mixer`-Instanzen zu bekommen, bietet die Klasse `AudioSystem` die statischen Methoden `getMixerInfo()` (zur Auflistung aller installierten Mixer) sowie `getMixer(Mixer.Info)` an.

Aufnehmen bzw. Abspielen von Audiodaten ist also lediglich über Subinterfaces von `DataLine` möglich: Das Aufnehmen erfolgt mit `TargetDataLine`-Instanzen, das Abspielen kann über `SourceDataLine`- oder `Clip`-Instanzen geschehen. Der Unterschied von `SourceDataLine` zu `Clip` besteht darin, dass ein `Clip` die abzuspielenden Daten zu Beginn komplett in den Speicher lädt, während `SourceDataLine` für kontinuierliches Streamen ausgelegt ist.

---

[140] Aus [JAVADOC].

Um `Port`- oder `DataLine`-Instanzen eines Geräts zu verwalten, existiert das `Mixer`-Interface. Eine `Mixer`-Instanz steht für eine Hard- oder Softwareeinheit, die je eine oder mehrere Eingangs- und Ausgangs-`Line`s besitzt. Das `Mixer`-Interface bietet unterschiedliche Methoden, um Informationen über an diesem `Mixer` verfügbare `Line`s zu bekommen oder auf diese zuzugreifen.

Zur Identifizierung einer `Line` dient die innere Klasse `Line.Info`. Ebenso besitzen die Subinterfaces von `Line` eigene Info-Klassen, genau genommen `Port.Info`, `Mixer.Info` und `DataLine.Info`. `Mixer.Info` erbt als einzige `Info`-Klasse nicht von `Line.Info`, was etwas verwundert, da `Mixer` von `Line` erbt.

Abbildung 13 zeigt diese nicht sofort schlüssig wirkende Interfacehierarchie inklusive der `Info`-Klassen. Die Vererbungshierarchie ist farblich hervorgehoben. Zu jedem Interface werden nur die wichtigsten Methoden – jeweils ohne Übergabe- und Rückgabeparameter – angegeben.

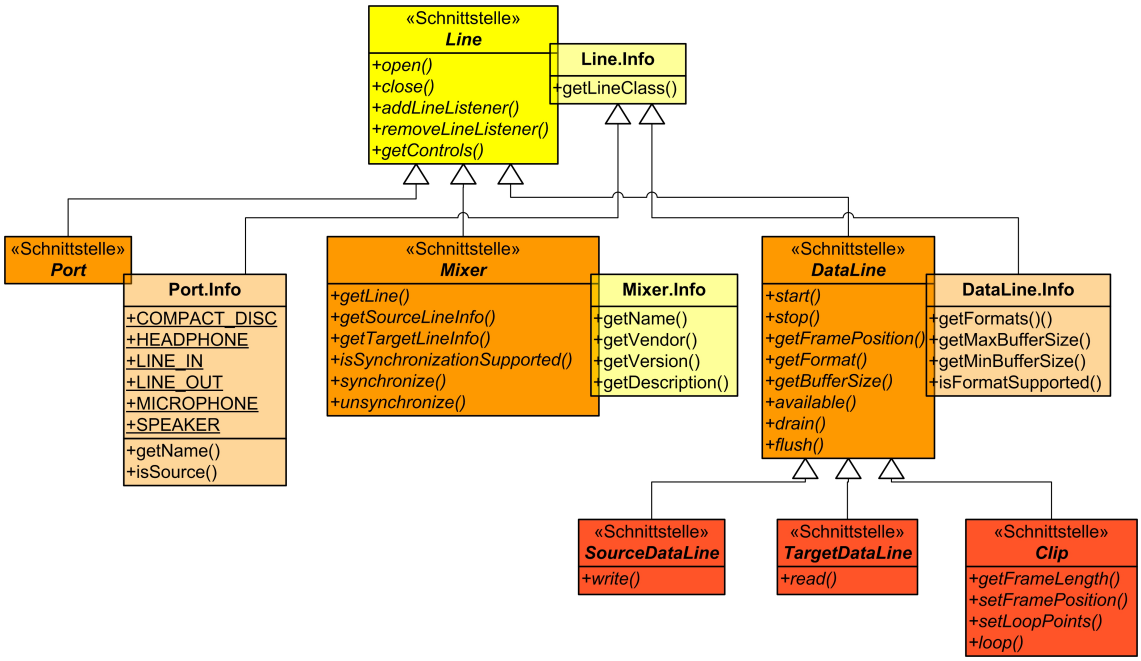

Abbildung 13: UML-Diagramm der `Line`-Interface-Hierarchie im `sampled`-Package

Die hier gewählte Namensgebung ist zumindest gewöhnungsbedürftig. So wundert man sich zunächst, warum man eine `SourceDataLine` zum Abspielen und eine `TargetDataLine` zum Aufnehmen benötigt. Jeder Programmierer würde wohl zunächst vom umgekehrten Fall ausgehen. Zum besseren Verständnis kann es hilfreich sein, sich vor Augen zu führen, dass bei der Namensgebung der `Mixer` im Mittelpunkt steht. Aus der Sicht des `Mixer`s ist eine `TargetDataLine` ein Zielobjekt, an das er Audiodaten schicken kann, und eine `SourceDataLine` eine Audioquelle, aus der er von der Applikation bereitgestellte Daten lesen kann.

```
1   // Mixer instanziieren
2   Mixer mixer = ...
3   try {
4     // Stream aus Quelldatei erzeugen
5     AudioInputStream stream =
6       AudioSystem.getAudioInputStream(new File("..."));
7     // Line zum Abspielen besorgen und öffnen
8     DataLine.Info info =
9       new DataLine.Info(SourceDataLine.class, stream.getFormat());
10    SourceDataLine sourceLine = (SourceDataLine)mixer.getLine(info);
11    sourceLine.open(stream.getFormat(), 8192);
12    // Lesen der Audiodaten vom Stream und Schreiben auf die DataLine
13    int numBytesRead = 0;
14    byte[] buffer = new byte[8192];
15    while(numBytesRead != -1){
16      numBytesRead = stream.read(buffer);
17      if(numBytesRead > 0) sourceLine.write(buffer, 0, numBytesRead);
18      sourceLine.start();
19    }
20    // Warten, bis der Ausgabepuffer der Soundkarte geleert ist
21    sourceLine.drain();
22    // Freigeben der Ressourcen
23    sourceLine.stop();
24    sourceLine.close();
25  } catch (Exception e) {
26    // Fehlermeldung
27  }
```

Codebeispiel 3: Abspielen einer Audiodatei als Stream

Codebeispiel 3 zeigt, wie in Java Sound eine Audiodatei als Stream abgespielt wird: Zunächst wird ein `AudioInputStream`-Objekt (s. u.) benötigt (Zeilen 5-6). Im nächsten Schritt muss ein `DataLine.Info`-Objekt erzeugt werden, das die gewünschte `DataLine` identifiziert. In diesem Fall soll es sich um eine `DataLine` zum gestreamten Abspielen (`SourceDataLine.class`) handeln, die das Format der Audiodatei (`stream.getFormat()`) unterstützt (Zeilen 8-9). Dieses Objekt wird nun in Zeile 10 dem `Mixer` übergeben, um von ihm eine passende `SourceDataLine`-Instanz zum Abspielen zu erhalten. Beim Öffnen der `SourceDataLine` (Zeile 11) kann das gewünschte Format sowie die Puffergröße angegeben werden. In den Zeilen 15-19 werden nun in einer Schleife Audiodaten aus der Datei in einen temporären Puffer und von dort auf die `SourceDataLine` geschrieben und damit abgespielt, und zwar so lange, bis das Dateiende erreicht ist und die `read()`-Methode `-1` zurückgibt. Das starten der `SourceDataLine` in Zeile 18 wird nur beim ersten Schleifendurchlauf benötigt. Es empfiehlt sich aber, die `SourceDataLine` erst zu starten, nachdem sich schon Daten in deren Puffer befinden, um einen buffer underrun zu Beginn des Abspielvorgangs zu verhindern. Nach dem Erreichen des Dateiendes wird die Methode `drain()` aufgerufen (Zeile 21), die erst zurückkehrt, wenn sämtliche im Puffer der `SourceDataLine` befindlichen Daten tatsächlich abgespielt wurden.

In Abbildung 14 wird die Bedeutung der gerade vorgestellten Interfaces im Audiodatenfluss veranschaulicht.

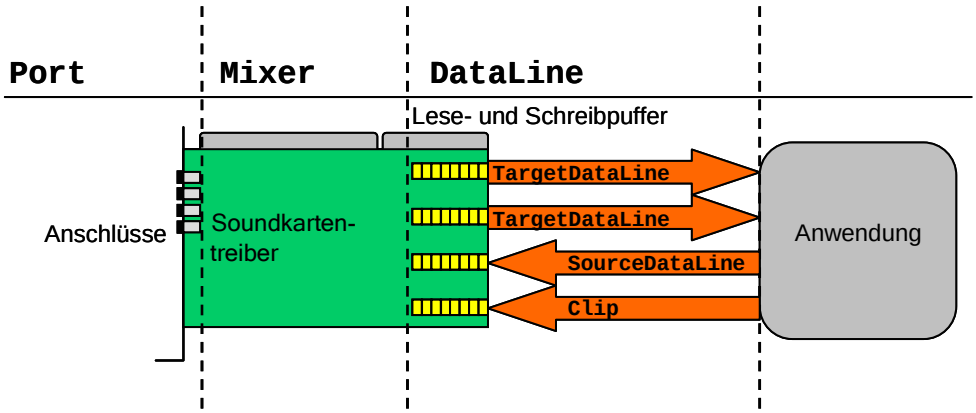

Abbildung 14: Bedeutung der `Line`-Interfaces im Audiodatenfluss

Controls

Um auf eventuell von Soundkartenherstellern zur Verfügung gestellte Besonderheiten wie beispielsweise eigene Effektsektionen, Equalizer oder einfache Lautstärkeregler zugreifen zu können, existiert in Java Sound die abstrakte Klasse `Control`. Jede `Line` – also auch jeder `Port`, jeder `Mixer` und jede `DataLine` – verfügt über die Methoden `isControlSupported(Control.Type)`, `getControls()` sowie `getControl(Control.Type)`, um Zugriff auf die für diese `Line` verfügbaren `Controls` zu bekommen.

Zur abstrakten Klasse `Control` existieren vier ebenfalls abstrakte Subklassen, um verschiedene Arten von `Controls` zu ermöglichen: `BooleanControl` (Auswahl aus zwei möglichen Werten), `FloatControl` (Angabe eines kontinuierlichen Wertes), `EnumControl` (Auswahl aus einer Liste von Werten) und `CompoundControl` (Enthält mehrere andere `Control`-Instanzen).

```
1   // Mixer instanziieren
2   Mixer mixer = ...
3   if(mixer.isLineSupported(Port.Info.LINE_OUT)){
4       // Port-Instanz vom Mixer holen
5       Port lineOut = (Port)mixer.getLine(Port.Info.LINE_OUT);
6       try{
7           if(lineOut.isControlSupported(FloatControl.Type.VOLUME)){
8             // Volume-Control besorgen
9             FloatControl volume =
10              (FloatControl)lineOut.getControl(FloatControl.Type.VOLUME);
11            // Lautstärke auf Maximalwert setzen
12            volume.setValue(volume.getMaximum));
13          }
14      } catch(LineUnavailableException e){
15          // Fehlermeldung
16      }
17  } else{
18      // Fehlermeldung
19  }
```

Codebeispiel 4: Verändern der Ausgangslautstärke des Line-Out-Anschlusses

Codebeispiel 4 zeigt, wie mit `Control`s die Ausgangslautstärke des Line-Out-Anschlusses verändert werden kann: Wie in Codebeispiel 2 wird bis Zeile 5 zunächst das `Port`-Objekt ermittelt, das den Line-Out-Anschluss des `Mixer`s repräsentiert. Das gewünschte Lautstärke-`Control` wird über das statische Feld `FloatControl.Type.VOLUME` identifiziert. Unterstützt das `Port`-Objekt diesen Typ von `Control` (Zeile 7), kann darauf zugegriffen und der Wert geändert werden (Zeilen 9-12).

Formate

In Java Sound werden die Sampledaten stets als `byte`-Array weitergegeben, so wie sie in Dateien gespeichert oder im Speicher abgelegt werden. Um diese Rohdaten interpretieren zu können, muss ihr Format bekannt sein, welches mit Objekten der Klasse `AudioFormat` definiert wird. `AudioFormat` enthält Informationen über Kodierung, Endianess[141], Zahl der Kanäle, Samplegröße, Samplingrate, Framegröße und Framerate. In unkomprimierten Formaten besteht ein Frame aus je einem Sample pro Kanal; die Parameter Samplegröße, Kanäle und Framegröße sowie Samplingrate und Framerate sind daher bei unkomprimierten Formaten voneinander abhängig.

Um Audiodaten von einem Format in ein anderes zu konvertieren, bietet die Klasse `AudioSystem` statische Methoden an, falls ein entsprechender `FormatConversionProvider` (siehe 2.3) installiert ist. Hierfür muss ein `AudioInputStream` (s. u.) im Ausgangsformat vorliegen. Mit `AudioSystem.isConversionSupported(AudioFormat, AudioFormat)` kann ermittelt werden, ob die gewünschte Konvertierung möglich ist, und mit `AudioSystem.getAudioInputStream(AudioFormat, AudioInputStream)` kann ein neuer `AudioInputStream` im gewünschten Format erzeugt werden.

Neben dem Format der Audiodaten selbst enthält die Klasse `AudioFileFormat` Eigenschaften von Audio-Dateiformaten, nämlich Dateityp, Dateilänge sowie die Länge der enthaltenen Audiodaten in Sample Frames.

Ab Version 1.5.0 bieten sowohl `AudioFormat` als auch `AudioFileFormat` mit den Methoden `getProperty(String key)` und `getProperties()` Zugriff auf zusätzliche Informationen, die in einem Format oder einer Datei enthalten sein können. [JAVADOC] macht hierzu Vorschläge für mögliche property keys.

Streams

Zur Verarbeitung von Audiodatenströmen bietet Java Sound die Klasse `AudioInputStream` an, welche von `java.io.InputStream` erbt. Sie erweitert `InputStream` um die Angabe eines `AudioFormat`s sowie die Gesamtlänge der Sampledaten. Diese kann aber auch den Wert `AudioSystem.NOT_SPECIFIED` annehmen, was bedeutet, dass die Länge unbekannt oder variabel ist. Ein weiterer Unterschied zu herkömmlichen `InputStream`s ist, dass das Lesen nur in Blöcken von ganzen Frames möglich ist. Hierdurch wird sichergestellt ist, dass der Beginn eines gelesenen Blocks immer mit dem Beginn eines Sample Frames zusammenfällt. Zum Erzeugen eines `AudioInputStream`s existieren zahlreiche Möglichkeiten, entweder über einen der beiden Kon-

[141] Unter *Endianess* (*big endian* oder *little endian*) versteht man die Reihenfolge der Bytes innerhalb eines Samples. Bei big endian wird das Most Significant Byte (MSB) an erster Stelle gespeichert, bei little endian dagegen das Least Significant Byte (LSB) (vgl. [JSRESOURCES]).

struktoren oder durch Aufruf einer der von `AudioSystem` zur Verfügung gestellten statischen Methoden.

Eine Parallele für das Schreiben von Audiodaten – man würde einen `AudioOutputStream` erwarten – existiert nicht. Das Schreiben von Audiodateien kann zwar der Methode `AudioSystem.write()` überlassen werden, diese genügt jedoch bei weitem nicht für alle denkbaren Anwendungsfälle, da sie nur einen Pull-Ansatz ermöglicht. Oft wird aber ein Push-Ansatz gewünscht, um detaillierter auf den Schreibprozess einwirken zu können. Das Open-Source-Projekt Tritonus bietet hierfür mit der `AudioOutputStream`-Architektur eine Alternative an. Auch im praktischen Teil dieser Arbeit wurde ein ähnlicher Ansatz gewählt (siehe VI.1.1.4).

2.2. Das Package javax.sound.midi

Im Vergleich zum `sampled`-Package ist das `midi`-Package einfacher und schlüssiger aufgebaut. Dies ist nicht weiter verwunderlich, da die MIDI-Verarbeitung weniger komplex ist als die Sampling-Verarbeitung.

MIDI-Geräte

Für die Repräsentation von MIDI-Geräten (Hard- oder Software) bietet Java Sound das `MidiDevice`-Interface, für das auch eine `MidiDevice.Info`-Klasse zur Identifizierung unterschiedlicher Geräte existiert. `MidiDevice` besitzt die beiden Subinterfaces `Sequencer` und `Synthesizer`.

Ein `Sequencer` bietet Methoden zum Abspielen und Aufnehmen von MIDI-Sequenzen auf mehreren Spuren (Klassen `Track` und `Sequence`), zur Beeinflussung des Tempos sowie zur Synchronisation und Änderung der Abspielposition.

Ein `Synthesizer` repräsentiert einen Software-Klangerzeuger, der MIDI-Nachrichten empfangen und dementsprechend Klänge wiedergeben kann. Über `Soundbank`-, `Instrument`- und `Patch`-Objekte können bestimmte Klänge in den `Synthesizer` geladen werden, die dann jedem `MidiChannel`-Objekt des `Synthesizer`s zugeordnet werden können. Um die bei softwareemulierten Klangerzeugern anfallenden Latenzen zu ermitteln, die zwischen dem Eintreffen der MIDI-Nachricht und dem Erzeugen des Klangs vergehen, bietet das `Synthesizer`-Interface die Methode `getLatency()`, die die im schlimmsten Fall zu erwartende Latenz angibt. Diese kann beim Abspielen berücksichtigt werden.

MIDI-Nachrichten

Für MIDI-Nachrichten enthält Java Sound die abstrakte Klasse `MidiMessage`, zu der drei konkrete Subklassen für unterschiedliche MIDI-Nachrichtentypen implementiert sind: `MetaMessage` steht für Meta-Nachrichten, die zwar von keinem Klangerzeuger interpretiert werden können, aber Informationen für den Sequenzer enthalten. Dies können z.B. Tempo- oder Taktartänderungen sein. Die gängigsten MIDI-Nachrichten wie Note On/Off, Controller, etc. (siehe III.1.2.2) werden durch die Klasse `ShortMessage` abgedeckt. Für systemexklusive (SysEx) Nachrichten steht schließlich noch die Klasse `SysexMessage` zur Verfügung.

Um MIDI-Nachrichten mit einem Zeitstempel zu versehen, um sie beispielsweise in einer Sequenz zu einem bestimmten Zeitpunkt abzuspielen, müssen `MidiMessage`-Objekte zusammen mit dem Zeitstempel in ein `MidiEvent`-Objekt gepackt werden.

```
1   // MidiDevice instanziieren
2   MidiDevice device = ...
3   try{
4     device.open();
5     // MIDI-Nachrichten erzeugen
6     ShortMessage noteOn = new ShortMessage();
7     noteOn.setMessage(ShortMessage.NOTE_ON, 0, 60, 100);
8     ShortMessage noteOff = new ShortMessage();
9     noteOff.setMessage(ShortMessage.NOTE_OFF, 0, 60, 0);
10    // Ausgangskanal öffnen und Nachrichten senden
11    Receiver output = device.getReceiver();
12    output.send(noteOn, -1);
13    Thread.sleep(1000);
14    output.send(noteOff, -1);
15    // Ressourcen freigeben
16    output.close();
17    device.close();
18  } catch(Exception e){
19    // Fehlermeldung
20  }
```

Codebeispiel 5: Senden von MIDI-Nachrichten mit Java Sound

In Codebeispiel 5 wird gezeigt, wie in Java Sound eine Note-On/Note-Off Sequenz gesendet werden kann: Dazu werden in den Zeilen 6-9 zunächst die entsprechenden `ShortMessage`-Objekte erzeugt und mit dem gewünschten Nachrichteninhalt gefüllt (Der Wert `0` entspricht hier dem ersten Kanal, `60` entspricht der Taste C1 und `100` ist die Anschlagsgeschwindigkeit). Beim Senden der Nachrichten an den `Receiver` des MIDI-Out-Anschlusses (Zeilen 11-14) muss ein Zeitstempel angegeben werden. Der Wert `-1` bedeutet hier ein sofortiges senden. Da die Note eine Sekunde lang klingen soll, wird in Zeile 13 eine Sekunde gewartet, bevor das Note-Off-Event gesendet wird.

MIDI-Verbindungen

Um interne MIDI-Verbindungen herzustellen, verwendet Java Sound ein Modell mit den Interfaces `Transmitter` und `Receiver`. Einem `Receiver` können über die Methode `send(MidiMessage)` MIDI-Nachrichten geschickt werden. Ein `Transmitter` sendet dagegen von sich aus MIDI-Nachrichten. Über `setReceiver(Receiver)` kann ihm ein `Receiver` zugeordnet werden, an den er die Daten schickt. Ohne `Receiver` sendet der `Transmitter` ins Leere. Jede `MidiDevice`-Instanz kann mit `getTransmitter()` oder `getReceiver()` um eine Instanz des gewünschten Interfaces gebeten werden. Verfügt die Instanz über keinen freien `Transmitter` bzw. `Receiver` mehr, wird eine `MidiUnavailableException` geworfen. Wird ein `Receiver` bzw. `Transmitter` nicht mehr benötigt, muss er explizit mit `close()` freigegeben werden, damit er wieder anderen Interessenten zur Verfügung steht.

2.3. Die SPI-Schnittstellen

Um Java Sound erweiterbar zu machen, was gerade in der sich schnell ändernden Audiowelt eine unverzichtbare Eigenschaft ist, wurden acht abstrakte Klassen in den Packages `javax.sound.sampled.spi` und `javax.sound.midi.spi` integriert. Diese können von Drittanbietern implementiert werden, um im Rahmen von Java Sound Zugriff auf bestimmte Geräte oder Formate zu ermöglichen, die standardmäßig von der gewählten Implementierung nicht unterstützt werden.

Folgende Services können als SPI von Drittanbietern zu Java Sound Implementierungen hinzugefügt werden:

- Lesen von Dateien (`AudioFileReader`, `MidiFileReader`, `SoundbankReader`)
- Schreiben von Dateien (`AudioFileWriter`, `MidiFileWriter`)
- Zur Verfügung Stellen von Soft- oder Hardwaregeräten (`MixerProvider`, `MidiDeviceProvider`)
- Formatkonvertierungen (`FormatConversionProvider`)

Um solche implementierten Services für andere Benutzer von Java Sound verfügbar zu machen, müssen diese zunächst in eine `.jar`-Datei gepackt werden. Innerhalb dieser Datei werden im Verzeichnis `META-INF/Services` die implementierten Services angegeben. Für jede abstrakte Klasse aus den beiden `spi`-Packages, für die eine oder mehrere Implementierungen erstellt wurden, muss eine Datei mit dem vollqualifizierten Namen dieser Klasse als Dateinamen (z.B. `javax.sound.sampled.spi.AudioFileReader`) in dieses Verzeichnis gelegt werden. Innerhalb dieser Datei werden dann die vollqualifizierten Klassennamen aller in der `.jar`-Datei enthaltenen Implementierungen dieses Services untereinander aufgelistet.

Der Inhalt dieser `.jar`-Datei muss nun noch in den Classpath eingefügt werden. Dies kann entweder dadurch geschehen, dass sie in das `/lib/ext`-Verzeichnis unterhalb des Java-Home-Verzeichnisses gelegt wird, oder dadurch, dass der Pfad zur `.jar`-Datei explizit dem Classpath hinzugefügt wird.

3. Theoretische Fähigkeiten und Grenzen von Java Sound

Zur Bewertung des Java Sound API (unabhängig von jeglicher Implementierung) werden im Folgenden die Anforderungen an professionelle Audiosoftware aus III.3 im Einzelnen durchgegangen. Dabei werden die theoretischen Fähigkeiten und Grenzen des Java Sound API in Bezug auf die jeweilige Anforderung herausgearbeitet.

Echtzeitfähigkeit

Java Sound bietet volle Kontrolle über echtzeitrelevante Daten. Hierzu zählt insbesondere die Wahl der Puffergrößen. Als Lowlevel-Framework überlässt es dem Anwendungsentwickler die Implementierung der echtzeitkritischen Bestandteile der Software und abstrahiert nur den Zugriff auf die Soundhardware. Einziger Punkt, in dem Java Sound negativ mit Echtzeitanforderungen in Berüh-

rung kommt, ist ein Designfehler im `midi`-Package: Die `MidiMessage`-Klasse ist veränderbar (mutable) gestaltet, so dass jede `MidiMessage` vor dem Versenden geklont werden muss. Die dadurch hervorgerufene hohe Zahl an Objektneuerzeugungen führt zu einer höheren Aktivität des Garbage Collectors und erschwert damit die Einhaltung von Echtzeitanforderungen.

Wünschenswert für eine Minimierung der Latenzen und somit ein besseres Echtzeitverhalten wäre außerdem eine Unterscheidungsmöglichkeit verschiedener Treibertypen bei der Auswahl von `Mixer`-Objekten (siehe Konfigurierbarkeit). So könnte dem Benutzer beispielsweise eine Liste der vorhandenen ASIO-Treiber zur Auswahl angeboten werden, was derzeit nicht möglich ist.

Unterstützung gängiger Formate und Standards

Welche Formate von einer auf Java Sound basierenden Software unterstützt werden, ist nicht abhängig vom Java Sound API. Durch das PlugIn-fähige Konzept von Java Sound (siehe Erweiterbarkeit) bleibt es den Entwicklern überlassen, welche Formate unterstützt werden. Auch sonst lässt sich kein Aspekt erkennen, in dem das Java Sound API für kommende Standards ungeeignet sein könnte.

Erweiterbarkeit

Mit dem Konzept der Service Provider Interfaces (siehe 2.3) ermöglicht Java Sound eine hohe Erweiterbarkeit für unterschiedliche Bereiche. So kann z.B. eine Unterstützung für das Lesen und Schreiben neuer Dateiformate ebenso von Drittanbietern hinzugefügt werden wie eine Anbindung an neue Treiberarchitekturen.

Allerdings ist der derzeitige Umgang mit den installierten SPI-Implementierungen durchaus noch verbesserungswürdig. Es kann beispielsweise nicht festgelegt werden, welche von mehreren installierten `AudioFileReader`-Implementierungen beim Aufruf von `AudioSystem.getAudioFileFormat()` bevorzugt verwendet wird.[142] In Version 1.5 wurden zwar ähnliche Mechanismen für andere Bereiche (z.B. `AudioSystem.getClip()`) eingeführt,[143] warum diese allerdings nicht auch auf z.B. `AudioFileReader/Writer` übertragen wurden, erscheint unverständlich. Vermisst wird auch die Möglichkeit, Provider zur Laufzeit hinzuzuladen oder zu entfernen.[144]

Ebenso wäre eine Art zentrale Registry – beispielsweise auf einer Website – für verfügbare Service Provider, Dateiformate (Klasse `AudioFileFormat.Type`), Kodierungstypen (Klasse `AudioFormat.Encoding`), property keys (z.B. für `AudioFileFormat.getProperty(String key)`) und Ähnliches wünschenswert. An dieser Stelle könnten sowohl vorhandene PlugIns, als auch darin verwendete Benennungen bekannt gemacht werden.

Ein Punkt, in dem Java Sound ebenfalls an die Grenzen seiner Erweiterbarkeit stößt, ist das Reagieren auf besondere Ereignisse zur Laufzeit. Viele Treibermodelle verfügen über so genannte Callback-Mechanismen. Der Treiber ruft hierbei zu beliebigen Zeiten bestimmte Callback-Funktionen auf, ähnlich den Listener-Interfaces im GUI-Bereich. Es wäre wünschenswert, dass Java Sound irgendeine Form der Registrierung und Benachrichtigung von Listener-Objekten ermöglichen wür-

142 Vgl. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5061439.
143 Vgl. [JAVASOUND], Appendix 2.
144 Vgl. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4666881.

de, damit Anwendungsentwickler auf Situationsänderungen zur Laufzeit reagieren könnten. Bisher besteht lediglich die Möglichkeit, auf das Öffnen Schließen, Starten oder Stoppen einer `Line`-Instanz zu reagieren. Eine mögliche Erweiterung könnte wie in Codebeispiel 6 aussehen: Statusänderungen könnten über die `propertyChanged()`-Methode bekannt gemacht werden und Fehlersituationen über `errorOccured()`.

```
1  /**
2   * Erweiterung des LineListener-Interface
3   */
4  public interface MixerListener extends LineListener{
5    /**
6     * Benachrichtigung über Änderung einer Eigenschaft
7     * @param key Der Schlüssel der geänderten Eigenschaft
8     * @param newValue Der neue Wert der Eigenschaft
9     */
10   public void propertyChanged(String key, Object newValue);
11   /**
12    * Benachrichtigung über das Auftreten eines Fehlers
13    * @param t Der aufgetretene Fehler
14    */
15   public void errorOccured(Throwable t);
16 }
```

Codebeispiel 6: Mögliche erweiterte Listener-Unterstützung für Java Sound

Konfigurierbarkeit

Auch die Konfigurierbarkeit scheint bei Java Sound auf den ersten Blick gut gelöst: Es existieren zu den wichtigen Klassen und Interfaces innere `*.Info`-Klassen, die zusätzliche Informationen über einzelne Objekte geben. Diese sind dann hilfreich, wenn dem Benutzer eine Liste von verfügbaren Objekten eines bestimmten Typs zur Auswahl angeboten werden soll. Auch können Format und Puffergröße für Aufnahme und Wiedergabe – im Rahmen der von Hardware und Treiber unterstützten Werte – bestimmt werden.

Allerdings offenbart das Java Sound API in punkto Konfigurierbarkeit auch einige Schwächen. So ist beispielsweise für `Mixer`-Objekte keine Unterscheidung nach deren Typ vorgesehen. Für professionelle Anwendungen ist es jedoch sehr wichtig, unterschiedliche Treiber nach den von ihnen verwendeten Treibermodellen unterscheiden zu können. Eine `Mixer.Type`-Klasse könnte sich hier in das Java Sound API gut einfügen. Bisher steht dagegen nur die `Mixer.Info`-Klasse zur Verfügung, die außer Name, Hersteller, Version und Beschreibung keine weiteren Informationen über ein `Mixer`-Objekt enthält. Ansonsten können lediglich aus den vom `Mixer` angebotenen `Line`s Rückschlüsse auf den Typ des `Mixer`s gewonnen werden – eine unzureichende Entscheidungshilfe.

Viele Audiotreiber bieten außerdem einen eigenen Konfigurationsdialog an, der wichtige Einstellungen enthalten kann. Es wäre durchaus denkbar, Methoden wie `Mixer.hasControlPanel()` und `Mixer.showControlPanel()` in das Java Sound API mit aufzunehmen, um Anwendungsentwicklern Zugriff auf diese Dialoge zu gewähren.

Nichtdestruktives Arbeiten

Als Lowlevel-Framework kommt Java Sound mit der Anforderung nach Möglichkeiten für Nichtdestruktives Arbeiten nicht in Berührung. Sämtliche unter III.1.3.2.3 vorgestellten Ansätze müssen vom Programmierer selbst implementiert werden.

Skalierbarkeit

Java Sound ist weitestgehend unabhängig von verwendeten Datenmengen gehalten. Das Arbeiten mit samplingbasierten Daten erfolgt über `AudioInputStream`s und damit unabhängig von der Gesamtdatenmenge. Wann und in welchem Umfang Daten in den Speicher geladen werden sollen, bleibt damit dem Programmierer überlassen.

Der einzige Punkt, an dem das Java Sound API im Sampling-Bereich nicht mit Streaming arbeitet, ist bei der Verwendung von `Clip`s. Diese stellen eine Möglichkeit zum Abspielen kurzer Stücke dar, die dabei komplett in den Speicher geladen werden und somit auch geloopt abgespielt werden können. Dem Programmierer sollte bei der Arbeit mit Java Sound bewusst sein, dass der Einsatz von `Clip`s nur dort Sinn macht, wo mit überschaubaren Datenmengen gerechnet werden kann. Sollen große Datenmengen abgespielt werden, sollten `SourceDataLine`-Instanzen anstelle von `Clip`s verwendet werden.

Eine kleine Schwachstelle bei der Arbeit mit großen Datenmengen stellt die Methode `AudioSystem.write()` dar (siehe 2.1, Streams). Da diese Methode erst nach Beendigung des Schreibvorgangs zum Aufrufer zurückkehrt, sollte sie eigentlich in einem eigenen Thread aufgerufen werden, um den weiteren Programmablauf – insbesondere das GUI – nicht unnötig lang aufzuhalten. Der einfache Aufruf verleitet jedoch dazu, dies nicht zu tun. Das Anbieten von `AudioOutputStream`s als Gegenstück zu `AudioInputStream`s wäre hier sicherlich die elegantere Variante gewesen.

Nutzen vorhandener Ressourcen

Abgesehen von den unter Konfigurierbarkeit angesprochenen Problemen kann unter Java Sound weitestgehend auf alle verfügbaren Ressourcen zugegriffen werden. Schwierig wird es natürlich dann, wenn spezielle Ressourcen, wie z.B. zusätzliche DSP-Chips angesprochen werden sollen. Hier kommt Java ganz deutlich an seine Grenzen. In solchen Fällen müssten über JNI Sonderlösungen gefunden werden. Das stellt allerdings keinen besonderen Nachteil von Java Sound dar, da ein ähnlicher Aufwand auch bei Benutzung anderer Frameworks nötig wäre.

Synchronisation

Das Java Sound API bietet an mehreren Stellen Unterstützung für Synchronisation: So können `Sequencer`-Objekte mit den Methoden `setMasterSyncMode()` und `setSlaveSyncMode()` in bestimmte Synchronisations-Modi versetzt werden. Ebenso steht für `Mixer`-Objekte die Methode `synchronize()` zur Verfügung, um mehrere `DataLine`s zueinander zu synchronisieren.

Für die Berechnung der Latenzen der an einer Synchronisationskette beteiligten Elemente steht lediglich die Methode `getLatency()` des `Synthesizer`-Interfaces zur Verfügung. Im `sampled`-Package sucht man – beispielsweise bei `DataLine` oder `Mixer` – eine solche Methode vergeblich.

Weiterhin besteht keine Möglichkeit zur Einbindung externer Synchronisationsquellen. Professionelle Soundkarten bieten beispielsweise Wordclock-Eingänge an, an die ein Clock-Signal zur Syn-

chronisation angelegt werden kann. In Java Sound existiert derzeit kein Mechanismus für den Zugriff auf solche Einstellungen.

4. Stand der Entwicklung

4.1. Die Referenzimplementierung von Sun

Um ein grobes Bild über den Stand der Entwicklung von Suns Referenzimplementierung von Java Sound zu bekommen, ist es hilfreich, sich die Verbesserungen und Bugfixes anzusehen, die in J2SE 1.5.0 hinzugekommen sind:

> „Ports are now available on all platforms. MIDI device I/O is now available on all platforms. Optimized direct audio access is implemented on all platforms. [...] Java Sound no longer prevents the VM from exiting."[145]

Ganz offensichtlich war der Stand der Java Sound Implementierung bis Version 1.4.x unvollständig und fehlerbehaftet. So war es auf vielen Plattformen nicht möglich, die vorhandenen Geräte abzufragen; es konnte also meist nur auf die für viele Anwendungen völlig unzureichende Java Sound Audio Engine zurückgegriffen werden.

Da J2SE 1.5.0 zu dem Zeitpunkt dieser Arbeit (Mitte Juli 2004) immer noch nur als Beta-Version erhältlich ist und da es einige Zeit dauern wird, bis sich diese Version bei den Endbenutzern durchsetzt, muss sich Java Sound diese Punkte noch als Kritik gefallen lassen. Allerdings zeigt die Liste der Verbesserungen in 1.5.0 auch, dass im Bereich Java Sound viel getan wurde und die verfügbare Beta-Version zeigt auch, dass sich diese Mühe gelohnt hat.

Zwar lässt sich schwer einschätzen, wie gut die Implementierung derzeit ist, da die Beta-Version von J2SE 1.5.0 erst seit kurzem verfügbar ist. In der Java Sound Mailingliste ([JS-INTEREST]) und auch bei den Erfahrungen, die im Rahmen des praktischen Teils dieser Arbeit mit der neuen Version gesammelt wurden, entsteht jedoch der Eindruck, dass Java Sound durch diesen Versionssprung gerade „erwachsen" geworden ist: Die Implementierung von Sun funktioniert jetzt weitestgehend so, wie vom API vorgesehen und es wird versucht, die im System vorhandene Hardware bestmöglich zu unterstützen.[146]

Jedoch gibt es auch in Version 1.5.0beta noch einige Punkte, die nicht implementiert sind. So unterstützt Java Sound nach wie vor standardmäßig nur Audioformate bis 16 Bit Samplegröße, 2 Kanäle und Ganzzahlkodierung. Da in professionellen Umgebungen oft mit 24 oder 32 Bit, Mehrkanal und/oder Fließzahlkodierung gearbeitet wird, ist das nicht ganz nachvollziehbar. Allerdings lassen sich diese Funktionen relativ einfach per Service Provider Interface nachrüsten.

Ein wichtiges Kriterium, das auch noch nicht den Weg in Version 1.5.0 gefunden hat, ist die Implementierung der Methode `Mixer.synchronize()`, die für eine Synchronisation von Audiodatenströmen innerhalb eines Gerätes verantwortlich sein soll, was für professionelle Anwendungen ein sehr wichtiger Aspekt sein kann.

---

145 Aus [RELNOTES].

146 Z.B. konnte mit Einführung der Direct Audio Devices (ALSA auf Linux, Direct Sound auf Windows) die Latenz beim Abspielen und Aufnehmen wesentlich verringert werden.

4.2. Java Sound auf dem Macintosh

Für die Implementierung von Java Sound für Mac-Betriebssysteme ist der Hersteller Apple als Lizenznehmer von Sun verantwortlich. Dafür stehen ihm die Implementierungen für andere Plattformen als Referenz zur Verfügung. In der aktuellen Java-Version für Mac, 1.4.2, scheint man sich bei Apple stark an die Referenzimplementierung von Sun gehalten zu haben, da sie einen ähnlich lückenhaften Stand aufweist, wie die 1.4.x-Implementierungen von Sun. Offensichtlich wurde auch die Java Sound Audio Engine übernommen. Dieser Eindruck ergibt sich aus einigen Tests auf Mac OS X im Rahmen des praktischen Teils dieser Arbeit und aus den Beiträgen von Mac-Usern in der Java-Sound-Mailingliste ([JS-INTEREST]). Wann die Java Sound Implementierung auf dem Mac die Qualität erreicht, wie sie die Referenzimplementierung von Sun in Version 1.5.0 vorlegt, kann nicht gesagt werden.

Hier wird ein großer Nachteil von Java Sound hinsichtlich der nach wie vor führende Plattform im professionellen Audiobereich, Apple Macintosh, deutlich: Sun hat kaum Einfluss auf die Qualität der Java Sound Implementierung für diese Plattform. Zwar kann davon ausgegangen werden, dass Apple den von Sun vorgelegten plattformunabhängigen Code übernehmen wird, aber der plattformspezifische Teil muss von Apple implementiert werden. Der Vorteil der Plattformunabhängigkeit droht dadurch verloren zu gehen, dass Java Sound auf unterschiedlichen Plattformen unterschiedlich gut implementiert wird. Das derzeit geringe Interesse von Apple an einer sinnvollen Unterstützung von Java Sound zeigt sich auch daran, dass für das CoreAudio-Framework auf Mac OS X zwar eine Java-Schnittstelle bereitgestellt wird, diese aber mit Java Sound nichts zu tun hat.

Allerdings eröffnet diese Java-Anbindung in Kombination mit dem SPI-Konzept von Java Sound Drittanbietern die Möglichkeit, eine Anpassung von CoreAudio für Java Sound zu schreiben. Mit solch einer Anpassung könnte Java Sound auch auf dem Mac voll ausgereizt werden. Das *Plumstone*-Projekt[147] hat genau diesen Weg eingeschlagen und eine SPI-Implementierung für eine Anbindung an das CoreAudio-MIDI-System entwickelt. Würde eine solche Anbindung auch für den Sampling-Bereich von CoreAudio realisiert, sähe die Situation für Java Sound auf dem Mac bedeutend besser aus.

4.3. Tritonus

Neben der Implementierung von Sun existiert für die Linux-Plattform noch eine Java Sound Implementierung des Tritonus-Projekts. Vor JDK 1.5.0 bot sie eine echte Alternative zur Sun-Implementierung, da sie einige Schwächen der Sun-Implementierung nicht aufwies. Im Übrigen wurde die neue Sun-Implementierung von den Entwicklern des Tritonus-Projekts, Florian Bömers und Matthias Pfisterer erstellt.

Tritonus basiert auf der GNU LGPL[148] und kann daher auch von JVM-Implementierungen von Drittanbietern genutzt werden, die keine Lizenznehmer von Sun sind, wie beispielsweise das Kaffe-Projekt[149].

---

147 http://www.cems.uwe.ac.uk/~lrlang/plumstone/

148 Lesser General Public License, http://www.gnu.org/copyleft/lesser.html

149 http://www.kaffe.org/

5. Bestehende Projekte auf Basis von Java Sound

Die Anzahl größerer Projekte, die auf Java Sound basieren ist derzeit noch überschaubar und nur die wenigsten davon haben bereits einen stabilen Status erreicht. Es finden sich darunter auch kaum Anwendungen, die den Anspruch erheben, studiotauglich zu sein.[150]

Gerade aufgrund der erwähnten Schwächen der Java-Sound-Implementierung verwundert es nicht, dass diese Projekte fast ausschließlich nichtkommerziell sind und meist eigene Workarounds für die Mängel in der Implementierung einsetzen und somit das Java Sound API umgehen. Bestes Beispiel hierfür ist die Synthesizer-Editor und -Librarian Software *JSynthLib*[151], für die eine eigene MIDI-Wrapper-Architektur entwickelt wurde. Diese erfordert zwar eine eigene Implementierung für jede unterstützte Plattform, ist dann aber nicht mehr zwingend auf Java Sound angewiesen ist.

Es finden sich allerdings auch einige Projekte, die komplett auf das Java Sound API aufbauen und entweder die derzeitigen Einschränkungen in Kauf nehmen oder durch den Einsatz von Service Provider Interfaces benötigte Funktionalitäten für Java Sound verfügbar machen, wobei diese Lösungen oft auf Kosten der Plattformunabhängigkeit gehen. Diese Projekte können von künftigen Verbesserungen der Java-Sound-Implementierung sowie von Service Provider Interfaces von Drittanbietern profitieren.

6. Alternativen zu Java Sound

Es fällt schwer, eine Konkurrenzsituation zu Java Sound aufzuzeigen, da Java Sound mit seinem Gesamtkonzept und als Bestandteil der Java-Plattform einmalig ist. Innerhalb von Java könnte man JMF bzw. MMAPI (siehe IV.1) als Konkurrenten ansehen, jedoch grenzt sich Java Sound durch seinen Lowlevel-Ansatz deutlich von diesen Highlevel-Fraemeworks ab. Davon abgesehen verwendet JMF intern für den Zugriff auf Soundhardware Java Sound. Die beiden Frameworks ergänzen sich also vielmehr als dass sie alternativ zueinander gesehen werden könnten. Trotzdem kann es auch im Studioumfeld Anwendungen geben, die nicht auf den hardwarenahen Ansatz von Java Sound angewiesen sind, dafür aber von den Highlevel-Features des JMF profitieren können.

Außerhalb der Java-Plattform finden sich noch ein paar Soundverarbeitungsframeworks, die in eine ähnliche Richtung zielen wie Java Sound. Am weitesten ausgereift ist wohl das Open-Source-Projekt *PortMusic* ([PORTMUSIC]), ein Lowlevel-Framework für Audioverarbeitung in C. Es besteht aus den Unterprojekten *PortAudio* (Sampling), *PortMidi* (MIDI) sowie *PortSoundFile* (Zugriff auf Audiodateien; derzeit noch in der Planungsphase). *„PortMusic is open-source and runs on Windows, Macintosh, and Linux."*[152] Damit ähnelt es in seinem Ansatz und Umfang dem Java Sound API.

Laut [PORTMUSIC] wird PortMusic schon von einigen bemerkenswerten Projekten eingesetzt, interessanterweise auch von Java-basierten. In punkto Zuverlässigkeit und Ausgereiftheit der Implementierung war PortMusic – abgesehen von PortSoundFile – Java Sound offensichtlich bis zuletzt überlegen. Mit Version 1.5 hat Java Sound allerdings einiges an Boden gut gemacht.

---

150 Eine stets aktuelle und umfangreiche Linksammlung von Java-Sound-Projekten findet sich auf [JSRESOURCES].

151 http://www.jsynthlib.org/

152 Aus [PORTMUSIC].

VI.Praktischer Teil

Ziel des praktischen Teils dieser Arbeit war es, eine Anwendung zu entwickeln, mit der die Studiotauglichkeit von Java Sound in einigen Bereichen überprüft werden kann. Entstanden sind zwei Versionen eines Sampleeditors sowie einige Java Sound PlugIns. Dort, wo Java Sound Schwächen offenbarte, wurde versucht, Alternativlösungen aufzuzeigen. Ein Großteil der praktischen Arbeit wurde in Form von wiederverwendbaren Paketen realisiert, die unabhängig von anderen Teilen arbeiten. Der gesamte Quellcode findet sich auf der beiliegenden CD im Verzeichnis `/Quellcode`, die Javadoc unter `/Javadoc`.

1. Die Beispielapplikationen

1.1. Java Sound PlugIns

Java Sound bietet mit den Service Provider Interfaces (siehe V.2.3) eine PlugIn-Architektur, um eine Unterstützung für zusätzliche Formate, Geräte oder Standards in Java Sound aufnehmen zu können. Um die Möglichkeiten dieser PlugIn-Schnittstelle zu evaluieren, wurden im Rahmen dieser Arbeit verschiedene PlugIns entwickelt, die im Folgenden vorgestellt werden.

Außerdem wird noch eine eigene PlugIn-Schnittstelle vorgestellt, die entwickelt wurde, um die Schwächen der `AudioFileReader/Writer`-SPIs zu umgehen.

1.1.1 ASIOMixerProvider

Beschreibung

ASIO (siehe III.2.3) ist heutzutage aus Studioumgebungen kaum noch wegzudenken. Insbesondere für das Erzielen von möglichst geringen Latenzen für besseres Echtzeitverhalten der Software hat sich ASIO als Standard etabliert. Keine Java-Sound--Implementierung bietet derzeit jedoch Zugriff auf ASIO-Treiber. Mit dem `ASIOMixerProvider`-PlugIn sollte eine Implementierung des `MixerProvider`-SPIs geschaffen werden, die das Ansprechen von ASIO-Treibern aus Java Sound heraus ermöglicht.

Ziele

Ziele waren hierbei im Einzelnen:

- Möglichst viele der von ASIO gebotenen Funktionalitäten so direkt wie möglich auf entsprechende Java-Sound-Komponenten abbilden.
- Den Overhead gering halten, um nicht den Hauptvorteil von ASIO (geringe Latenz) zu verlieren.
- Verwendung von nativem Code nur zum Zugriff auf die API-Funktionen von ASIO zur Wahrung bestmöglicher Plattformunabhängigkeit.

- ASIO-Bestandteile, die sich in Java Sound nicht abbilden lassen, als zusätzliche Methoden anbieten.

Insbesondere die Verwirklichung der ersten beiden Ziele erwies sich als schwierig, da die Grundkonzepte von Java Sound und ASIO unterschiedlicher kaum sein könnten. So baut ASIO auf ein Callback-Modell, in dem der Treiber ständig Nachrichten an die Anwendung schickt, wenn er neue Audiodaten benötigt oder wenn in einer Aufnahmesituation neue Daten zur Verarbeitung vorliegen. Java Sound sieht hingegen vor, dass die Anwendung laufend in den vom Treiber vorgegebenen Puffer schreibt oder aus ihm liest. Darüber hinaus finden sich noch viele weitere Punkte, in denen die Unterschiedlichkeit der Ansätze erkennbar wird. In Tabelle 2 sind die Hauptunterschiede aufgeführt.

Tabelle 2: Hauptunterschiede im Konzept von ASIO und Java Sound

	ASIO	Java Sound
Grundlegender Ansatz	Aufnahme: Push Abspielen: Pull (Callback-basiert)	Aufnahme: Pull Abspielen: Push
Verfügbarkeit	Höchstens ein ASIO-Treiber gleichzeitig von höchstens einer Anwendung nutzbar	Keine Einschränkung
Datenkanäle	Pro Treiber mehrere Mono-Kanäle mit einheitlichem Format	Pro Mixer mehrere `DataLines`; können unterschiedliche Kanalzahlen und Formate unterstützen
Öffnen/Schließen der Datenkanäle	Alle Kanäle auf einmal	Alle `DataLines` unabhängig voneinander
Starten/Stoppen der Datenkanäle	Alle Kanäle auf einmal	Alle `DataLines` unabhängig voneinander
Synchronisieren der Datenkanäle zueinander	Automatisch	Muss explizit aufgerufen werden
Organisation der Datenkanäle	Jeder Kanal hat eine eindeutige Bezeichnung; Kanäle sind in Gruppen organisiert.	Keine
Einheit für Puffergrößen	Sample frames (unabhängig von Format und Kanalzahl)	Bytes

Realisierung

Zunächst wurden für sämtliche Bestandteile des ASIO-SDK[153] Wrapper-Klassen in Java geschrieben, um das Zusammenspiel dieser Elemente komplett in Java realisieren zu können. Anschließend wurden Implementierungen für die Interfaces `Mixer`, `SourceDataLine` und `TargetDataLine` geschrieben, die deren Methoden in ASIO übersetzen. Da sich `SourceDataLine` und `TargetDataLine` lediglich durch die Methoden `read()` und `write()` unterscheiden und da in ASIO die Aufnahme und Wiedergabe von Audiodaten über den selben Mechanismus realisiert wird, wurden beide Interfaces in ein und derselben Klasse (`ASIODataLine`) implementiert.

[153] Software Development Kit

Der ASIO-Treiber gibt beim Vorbereiten zum Abspielen oder Aufnehmen zwei Speicheradressen zurück, die zwei Pufferhälften markieren. Zu jeder Zeit ist eine von beiden Pufferhälften die aktive, in die die Anwendung Audiodaten schreibt (Abspielen) oder aus der sie Audiodaten liest (Aufnahme). Nachdem der Aufnahme- oder Abspielvorgang gestartet wurde, sendet der ASIO-Treiber regelmäßig Callbacks, um anzuzeigen, dass die aktive Pufferhälfte gewechselt werden soll. Diese Funktionalität in Java zu übersetzen war nicht ganz trivial, zumal für jeden Callback der aktuelle Thread an eine JVM-Instanz „angehängt" werden muss. Dazu muss diese natürlich schon vorher bekannt sein.

Ein weiteres Beispiel für die Schwierigkeit, die sich bei der Umsetzung der ASIO-Bestandteile in Java Sound ergab, ist der Umgang mit den einzelnen Kanälen, die ein Treiber zur Verfügung stellt: Ein ASIO-Treiber hat eine feste Anzahl von Ein- und Ausgangskanälen mit einem einheitlichen Format, von denen für jeden Aufnahme- oder Abspielvorgang eine beliebige Zahl verwendet werden kann; allerdings kann zu jedem Zeitpunkt nur ein Aufnahme- oder Abspielvorgang aktiv sein. Anders bei Java Sound: Jede `Mixer`-Instanz stellt eine Anzahl von `Source`- und `TargetDataLine`s sowie `Clip`s zur Verfügung. Diese können unabhängig voneinander geöffnet, geschlossen, gestartet und gestoppt werden und sind vom Format her unabhängig voneinander Es ist auch möglich, dass eine `DataLine` auf mehreren Kanälen aufnimmt oder abspielt; in ASIO ist dagegen jeder Kanal ein Mono-Kanal. Daraus ergibt sich unter anderem die Frage, wie damit umzugehen ist, wenn ein Benutzer von einem `ASIOMixer` eine `DataLine` in Monoformat anfordert. Handelt es sich bei der Soundkarte um ein Stereogerät, kann davon ausgegangen werden, dass das Mono-Signal auf der linken und rechten Seite ausgegeben werden soll. Wird dagegen eine Mehrkanalsoundkarte angesprochen, ist es gut möglich, dass die Ausgabe nur auf dem ersten Kanal gewünscht ist. Im `ASIOMixerProvider`-PlugIn wurde die zweite Lösung gewählt. Das Öffnen einer `DataLine` im Mono-Format führt also dazu, dass nur auf dem ersten (bei Stereokarten dem linken) Kanal abgespielt oder aufgenommen wird. Dieses Verhalten wird Benutzer von Stereosoundkarten möglicherweise verwundern, aber für die Mehrkanalunterstützung scheint es der bessere Weg zu sein.

Ergebnis

Das fertige ASIO-PlugIn ermöglicht den Zugriff auf installierte ASIO-Treiber aus Java Sound heraus[154]. Derzeit existiert nur eine Windows-Implementierung. Für Mac müsste der native Teil leicht angepasst und neu kompiliert werden. Bei bestimmten ASIO-Treibern kann es nötig sein, für ein korrektes Funktionieren das `FloatConversionProvider-PlugIn` (siehe nächster Abschnitt) installiert zu haben.

Das Abspielen und Aufnehmen funktioniert, allerdings tritt bei kleinen Puffergrößen und bei hoher Systemlast Knacksen im Audiosignal auf, die sich wohl darauf zurückführen lassen, dass die Daten nicht schnell genug verarbeitet werden. Ob diese Probleme auf Java, Java Sound oder die Implementierung zurückzuführen sind, konnte nicht herausgefunden werden. Ein mögliches „Bottleneck" könnte das ständige An- und Abmelden des Callback-Threads bei der JVM sein, das bei kleinen Puffergrößen sehr häufig ausgeführt werden muss.

---

[154] Um den `ASIOMixerProvider` auf einem Windows-System ohne vorhandenen ASIO-Treiber zu testen, kann der ASIO4All-Treiber (siehe III.2.3) installiert werden.

Weiterhin konnten nicht alle Bestandteile des ASIO-SDK wunschgemäß in Java Sound übersetzt werden. Es wurden einige Zusatzelemente wie beispielsweise ein `ASIOListener` eingeführt, auf die man aber naturgemäß nur zugreifen kann, wenn man das `Mixer`-Objekt explizit auf `ASIOMixer` castet und die Aufrufe kennt (siehe Javadoc).

Das `ASIOMixerProvider`-PlugIn besteht aus den Klassen im Package `com.groovemanager.spi.asio`. Der Quellcode für den nativen Teil findet sich im Verzeichnis `/Quellcode/Nativ/jsasio`. Um den nativen Teil kompilieren zu können, muss das ASIO-SDK von Steinberg[155] heruntergeladen und eingebunden werden. Um das PlugIn nutzen zu können, muss die native Bibliothek zusätzlich zum `.jar`-Archiv installiert werden. Beide liegen unter `/PlugIns/jsasio`.

1.1.2 FloatConversionProvider

Beschreibung

In den meisten gängigen Audioapplikationen werden heutzutage die Audiodaten intern als 32 Bit Fließkommazahlen – normalisiert auf +/- 1,0 – verarbeitet. Auch gibt es immer mehr Treiber, die diese Form der Kodierung der Audiodaten wählen. Die Vorteile liegen auf der Hand: Zum einen findet die Verarbeitung der Audiodaten in Effekten meist ohnehin in Fließkommaform statt, da sich damit einfacher mathematische Berechnungen durchführen lassen. Zum anderen erhält man durch die Normalisierung auf den Maximalwert 1,0 einen Headroom, der hilft, digitale Übersteuerungen bei der Verarbeitung zu vermeiden. Das Gegenargument, dass Fließkommaberechnungen langsamer sind als Ganzzahlberechungen, fällt bei heutigen CPUs mit ausgewachsenen Fließkommaeinheiten nicht mehr entscheidend ins Gewicht.

Daher verwundert es schon, dass Java Sound derzeit keine explizite Unterstützung dieses Formats bietet. Als mögliche vorgegebene Kodierungen existieren `PCM_SIGNED`, `PCM_UNSIGNED`, `ALAW` sowie `ULAW`. Die ersten beiden bezeichnen PCM-kodierte Ganzzahlen und die letzten beiden sind spezielle Kodierungsverfahren.

Um eine standardisierte Verarbeitung von Audiodaten als Fließkommazahlen zu ermöglichen, wurde in dieser Arbeit ein `FloatConversionProvider`-PlugIn entwickelt, das auch an anderen Stellen dieses Praxisteils Verwendung findet.

Ziele

Hauptziel bei der Entwicklung des `FloatConversionProvider`-PlugIns war, es so zu schreiben, dass es sich nahtlos in bestehende Java Sound Umgebungen einfügen kann. Ebenso wurde Wert auf eine effiziente Implementierung gelegt, da die Formatkonvertierungen oft in Echtzeit erfolgen müssen.

Realisierung

Zur Realisierung des PlugIns musste ein neuer Kodierungstyp in Java Sound eingeführt werden. Der Name `PCM_FLOAT` soll dem Umstand Rechnung tragen, dass die Kodierung nach wie vor in PCM-Form stattfindet, jedoch die Haltung der kodierten Daten im Float-Format geschieht.

---

[155] http://www.steinberg.de/Steinberg/Developers.asp?Langue_ID=4

Weiterhin wurde eine Subklasse von `InputStream` implementiert, die die eigentliche Konvertierung übernimmt. Abbildung 15 zeigt am Beispiel der Konvertierung von 16 Bit Integer little endian zu 32 Bit Float big endian, wie die Konvertierung gelöst wurde:

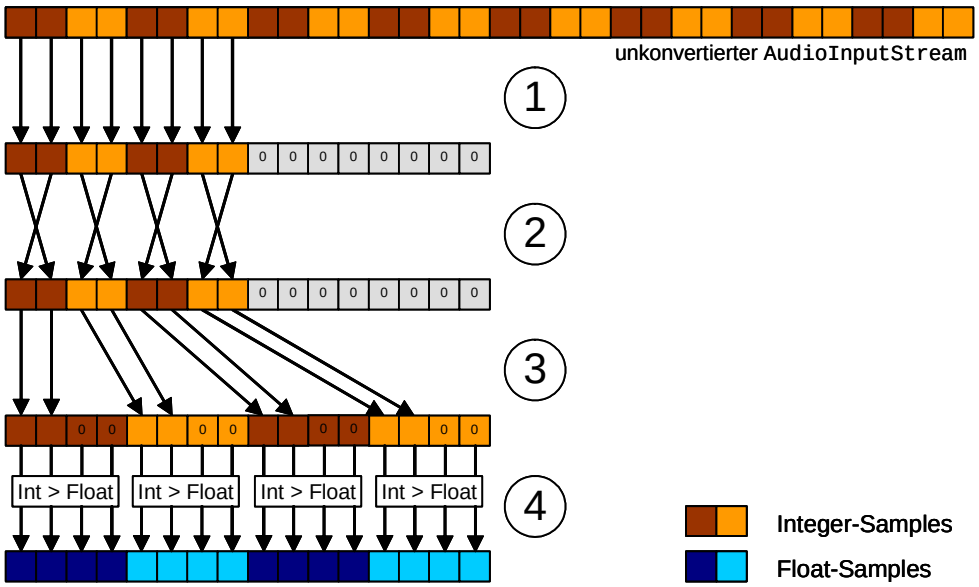

Abbildung 15: Ablauf der Konvertierung von 16 Bit Integer little endian zu 32 Bit Float big endian

Zunächst wird die benötigte Datenmenge vom unkonvertierten `AudioInputStream` gelesen (1). Anschließend werden die Bytes innerhalb jedes Samples umgekehrt gereiht, um von little endian zu big endian zu konvertieren (2). Im nächsten Schritt werden 0-Bytes eingefügt, um auf die Samplegröße von 32 Bit zu kommen (3), wonach im letzten Schritt die Konvertierung jedes Samples von Integer in Float erfolgt (4).

Abschließend wurde noch das eigentliche PlugIn, eine Subklasse des SPI `FormatConversionProvider`, implementiert, die es ermöglicht, über die definierten API-Schnittstellen von Java Sound den Service der Konvertierung in Anspruch zu nehmen.

Ergebnis

Der `FloatConversionProvider` konnte mit den gewünschten Fähigkeiten realisiert werden. Das fertige PlugIn kann in alle auf Java Sound basierende Systeme auf sämtlichen Plattformen eingebunden und dort genutzt werden. Da allerdings keine zentrale Stelle existiert, die mögliche Namen für Kodierungsarten in Java Sound definiert, muss dem Entwickler, der das PlugIn einsetzen möchte, der gewählte Name `PCM_FLOAT` für die Fließkommakodierung und dessen Bedeutung bekannt sein.

Alle benötigten Klassen sind im Package `com.groovemanager.spi.floatEncoding` enthalten. Die kompilierte Version befindet sich auf der beiliegenden CD-Rom im Verzeichnis `/PlugIns/floatconversion`.

1.1.3 REXFileReader

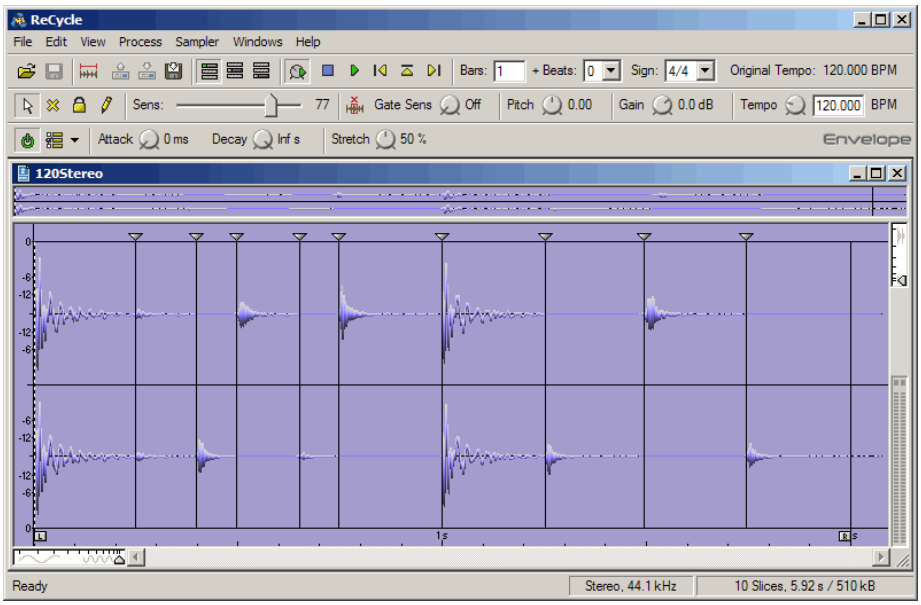

Abbildung 16: Die Loop-Software Recycle

Beschreibung

Das REX-Dateiformat[156] wurde für die Loop-orientierte Audiosoftware *Recycle*[157] (siehe Abbildung 16) entwickelt und erfreut sich im kreativen Audiobereich großer Beliebtheit. Es beinhaltet die Audiodaten eines Loops und darüber hinaus Informationen über die zeitliche Abfolge dessen einzelner Bestandteile (diese werden als *Slices* bezeichnet). So können diese unabhängig voneinander weiterverarbeitet werden, was neue Möglichkeiten im Arbeiten mit Loops eröffnet.

Die Firma *Propellerheads*, Entwickler dieses Formats, bietet ein SDK für die Programmiersprache C an, mit dem es möglich ist, REX-Dateien zu lesen, allerdings nicht zu schreiben. Um Java-Entwicklern Zugriff auf dieses Format zu ermöglichen, wurde im Rahmen dieser Arbeit ein `REXFileReader`-PlugIn für Java Sound entwickelt.

Ziele

Das Ziel bei der Entwicklung des `REXFileReader`s war, das REX-SDK in Java Sound zu abstrahieren und damit aus Java Sound heraus Zugriff auf REX-Dateien und deren Eigenschaften zu ermöglichen. Der `REXFileReader` sollte in der Lage sein, neben den reinen Audiodaten auch alle weiteren in der jeweiligen Datei enthaltenen Informationen zu lesen und weiterzugeben.

Realisierung

Ähnlich wie beim `ASIOMixerProvider` wurden zunächst Wrapper-Klassen in Java geschrieben, um in Java Zugriff auf alle Bestandteile des REX-SDK zu bekommen. Im Gegensatz zum ASIO-SDK wirkt das REX-SDK wesentlich aufgeräumter und stimmiger, weshalb die Einarbeitung weniger schwierig war.

---

156 http://www.propellerheads.se/technologies/rex/
157 http://www.propellerheads.se/products/recycle/

Um die Audiodaten als `AudioInputStream` bereitstellen zu können, musste dann eine Subklasse von InputStream implementiert werden, die die SDK-Funktionen abstrahiert. Weiterhin musste festgelegt werden, welche Dateiformateigenschaften auf welche Weise in den properties der `AudioFileFormat`-Klasse dargestellt werden. Tabelle 3 zeigt die gewählte Zuordnung der REX-Dateieigenschaften zu property keys. Auf die in [JAVADOC] vorgeschlagenen property keys wurde dabei Rücksicht genommen.

Tabelle 3: REX AudioFileFormat properties

Property key	Datentyp	Beschreibung
`"duration"`	`Long`	Gesamtdauer der Audiodaten in Mikrosekunden
`"author"`	`String`	Name, Emailadresse und URL des Autors (optional)
`"copyright"`	`String`	Copyright-Informationen (optional)
`"comment"`	`String`	Kommentar (optional)
`"slice_count"`	`Integer`	Anzahl der Slices in dieser Datei
`"bpm"`	`Float`	Tempo in beats per minute
`"time_signature"`	`int[]`	Taktart: Zähler an Index `[0]` und Nenner an Index `[1]`
`"slices"`	`int[][]`	Die Slices in dieser Datei. Das Array enthält für jeden Slice ein Array der Größe 2 mit der Position des Slice in Sample Frames an Index `[0]` und der Länge des Slice in Sample Frames an Index `[1]`.

Als letzter Schritt musste noch eine Implementierung der `AudioFileReader` SPI-Klasse geschrieben werden, um den Service einer Java-Sound-Umgebung anbieten zu können. Hierbei ergab sich folgendes Problem: Das REX-SDK fordert, dass REX-Dateien komplett in den Speicher geladen werden müssen, um Informationen oder Audiodaten aus ihnen lesen zu können. Streaming ist nicht vorgesehen. Die `AudioFileReader`-Klasse bietet dagegen Methoden, um Audiodaten oder Formateigenschaften aus einer Datei, einem `InputStream` oder einer `URL` zu lesen. Dieser Umstand führt zu folgender Einschränkung: Erstens kann der `REXFileReader` nur bei Übergabe eines `File`-Objekts sinnvolle Ergebnisse liefern, nicht aber aus einem `InputStream` oder einer `URL`. Das zweite Problem besteht darin, dass der `REXFileReader` theoretisch jede ihm angebotene Datei komplett in den Speicher laden müsste, um herauszufinden, ob es sich um eine gültige REX-Datei handelt. Da dies bei sehr großen Dateien ineffizient ist oder sogar zu Out-of-Memory-Fehlern führen kann, wurde eine maximale Dateigröße von 4 MB für REX-Dateien im `REXFileReader` verankert.

Wird der `REXFileReader` also nach einem `AudioFileFormat` oder `AudioInputStream` aus einem `InputStream`, einer `URL` oder einer mehr als 4 MB großen Datei gefragt, muss er zwangsläufig eine `UnsupportedAudioFileException` werfen, obwohl es sein kann, dass es sich bei den gegebenen Daten um gültige REX-Inhalte handelt.

Ergebnis

Das fertige PlugIn funktioniert mit der erwähnten Einschränkung auf Windows Betriebssystemen. Da es das REX-SDK auch für Macintosh gibt, wäre eine Anpassung dafür auch denkbar. Hierfür müsste der native Teil des PlugIns auf einem Mac-System kompiliert und als JNI Bibliothek ver-

fügbar gemacht werden. Zusätzlich zu diesem PlugIn sollte auch das `FloatConversionProvider-PlugIn` installiert werden, da die Audiodaten im `PCM_FLOAT`-Format bereitgestellt werden.

Zusätzlich zur Installation des `.jar`-Archivs und der nativen Bibliothek muss für dieses PlugIn noch die REX Shared Library installiert werden. All diese Elemente finden sich im Verzeichnis `/PlugIns/jsrex` auf der beiliegenden CD. Die Klassen sind im Package `com.groovemanger.spi.rex` enthalten. Der Quellcode für den nativen Teil liegt unter `/Quellode/Nativ/jsrex`.

1.1.4 Eigene PlugIn-Schnittstelle

Beschreibung

Wie bereits unter V.3 beschrieben, weist die Umsetzung der `AudioFileReader`- und `AudioFileWriter`-SPIs Mängel auf, die deren Einsatz erschweren. Dies wurde auch im Rahmen der Arbeit am Sampleeditor deutlich. Daher entschied ich mich, alternativ dazu eine eigene PlugIn-Schnittstelle für das Ermitteln des `AudioFileFormat`s aus einer gegebenen Datei sowie für das Schreiben von Audiodaten zu entwickeln, die diese Schwächen nicht aufweist.

Ziele

Die Schnittstelle sollte folgende Möglichkeiten bieten:

- Anlehnung an das PlugIn-Konzept von Java Sound
- An- und Abmelden verfügbarer PlugIns
- Qualitative Unterscheidung der verfügbaren PlugIns
- Flexible Einsetzbarkeit

Realisierung

Als Basis für die Implementierung von PlugIns wurden zwei abstrakte Basisklassen entwickelt, von denen PlugIn-Implementierungen erben sollten: `AudioFileFormatProvider` für PlugIns zum Lesen des Dateiformats sowie `AudioFileOutputStreamProvider` für das Bereitstellen von `OutputStream`s zum Schreiben von Audiodatenströmen. Für diese `OutputStream`s wurde ebenfalls eine abstrakte Basisklasse `AudioFileOutputStream` geschaffen, die wiederum von `OutputStream` erbt.

Die Zentrale für die Verwaltung der PlugIns stellt – parallel zu `AudioSystem` in Java Sound – die Klasse `AudioManager` dar. Im Gegensatz zu `AudioSystem` sind die meisten Methoden jedoch nicht statisch, so dass mehrere `AudioManager`-Instanzen für unterschiedliche Einsatzwecke parallel genutzt werden können.

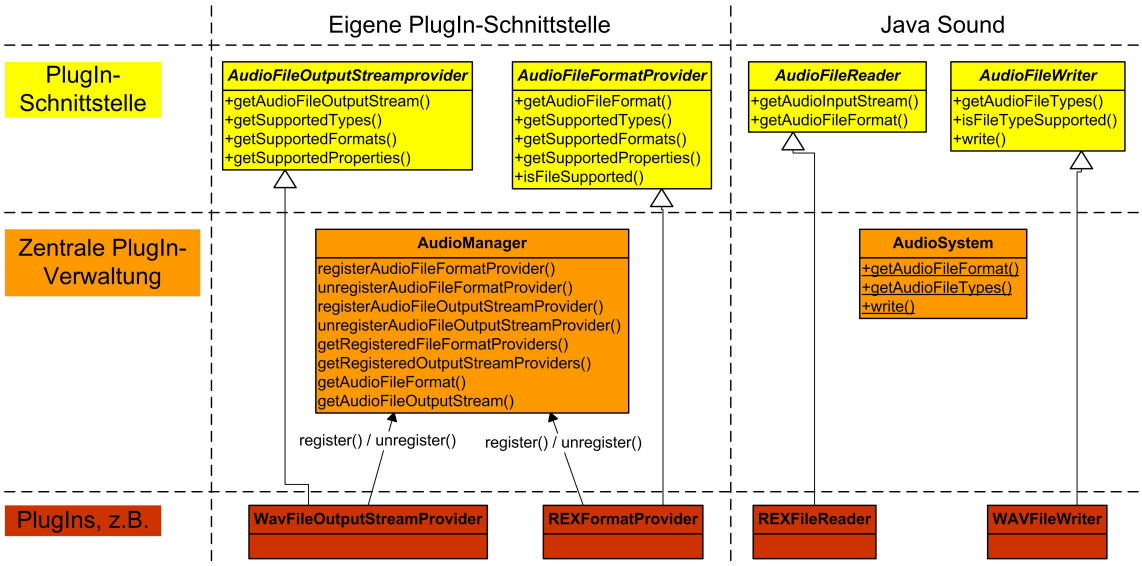

Abbildung 17: Vergleich der PlugIn-Schnittstellen

Instanzen vorhandener PlugIns können bei einer `AudioManager`-Instanz über die `register...Provider()` bzw. `unregister...Provider()`-Methoden an- bzw. abgemeldet werden. Somit kann kontrolliert werden, welche PlugIns von welcher `AudioManager`-Instanz verwendet werden. Um die unterschiedlichen PlugIns qualitativ unterscheiden zu können, existieren die Methoden `getSupportedTypes()` zur Bestimmung der les- bzw. schreibbaren Audiodateitypen, `getSupportedFormats()` zum Ermitteln der unterstützten Formate sowie `getSupportedProperties()`, um zu erfragen, welche property keys für `AudioFileFormat.getProperty(String key)` dem jeweiligen Provider bekannt sind. Letzteres wird insbesondere dann entscheidend, wenn mehrere PlugIns für den gewünschten Datei- und/oder Formattyp verfügbar sind, aber besondere Eigenschaften des Dateiformats genutzt werden sollen, die nicht von jedem PlugIn unterstützt werden.

Beim Aufruf der Methoden `getAudioFileFormat()` bzw. `getAudioFileOutputStream()` einer `AudioManager`-Instanz kann angegeben werden, welche property keys zwingend erforderlich sind und welche wünschenswert wären. Aus diesen Angaben wird der `AudioManager` dann den am besten passenden Provider ermitteln und den Aufruf an ihn weiterleiten. Wird kein Provider gefunden, der alle zwingend benötigten property keys unterstützt, so wird eine `UnsupportedAudioFileException` geworfen.

Ergebnis

Die entstandene PlugIn-Schnittstelle zeigt Wege auf, wie die SPI-Schnittstelle von Java Sound erweitert werden könnte, um die beschriebenen Schwächen zu überwinden. Sie wurde auch an verschiedenen Stellen im Sampleeditor eingesetzt, wo ein Einsatz der Java Sound SPI-Schnittstelle nicht ausreichend gewesen wäre. Alle dazugehörigen Klassen befinden sich im Package `com.groovemanager.sampled.providers`.

1.2. Sampleeditor

Hauptinhalt des praktischen Teils dieser Arbeit war die Entwicklung eines Sampleeditors auf Basis von Java Sound. Dieser wurde in zwei Versionen fertig gestellt: Die Grundversion enthält die Basisfunktionen eines Sampleeditors sowie eine visuelle Kontrolle über die verwendeten Ressourcen. Die MC-909 Version verzichtet auf diese visuelle Kontrolle, erweitert dafür aber die Grundversion um einige Funktionen für die Kommunikation mit der Hardware-Groovebox Roland MC-909.

Die Installation wird in Anhang C ausführlich beschrieben. Die spezifischen Klassen für die Grundversion finden sich im Package `com.groovemanager.app.sse`, die der MC-909-Version im Package `com.groovemanager.app.mc909se`. Alle weiteren Packages enthalten gemeinsam genutzte Elemente, die auch von anderen Anwendungen wiederverwendet werden könnten.

Beide Versionen sind unter Windows und Linux getestet worden. Theoretisch müssten sie auch auf jeder anderen Plattform mit SWT-Implementierung lauffähig sein.

1.2.1 Die Grundversion

Ein Sampleeditor dient zur Bearbeitung samplingbasierter Audiodateien. Dazu sollte er sowohl eine graphische Darstellung dieser Daten als auch elementare Grundfunktionen wie Öffnen, Speichern, Aufnehmen oder Abspielen unterstützen. Hinzu kommen Basisfunktionen zur Bearbeitung des Audiomaterials wie Löschen, Copy/Paste usw.

Beispiele für bekannte professionelle Sampleeditoren sind Steinbergs Wavelab, Adobes Audition (früher Cool Edit) oder SoundForge von Sonic Foundry.

Bei der Entwicklung der Grundversion des Sampleeditors sollten folgende Ziele erreicht werden:

- Kennen lernen und Testen des Java Sound `sampled`-Packages
- Orientierung an den Anforderungen an Audiosoftware im Studiobereich (siehe III.3)
- Implementierung elementarer Grundfunktionalitäten eines Sampleeditors
- Konsequenter Einsatz von Nichtdestruktiver Bearbeitung (siehe III.1.3.2.3)
- Entwicklung wiederverwendbarer Einzelkomponenten

Auf eine detaillierte Beschreibung der Implementierung wird an dieser Stelle mit Verweis auf den Quellcode und die Javadoc auf der beiliegenden CD verzichtet. Vielmehr werden die Bestandteile des Sampleeditors und ihre Funktion kurz vorgestellt. Abbildung 18 zeigt die Hauptbestandteile des Sampleeditors, auf die im Folgenden eingegangen wird:

1. File-, Edit-, Effect- und Options-Menü
2. Info-Bereich
3. Wellenformdarstellung
4. Transportleiste
5. Scroll- und Zoom-Bereich

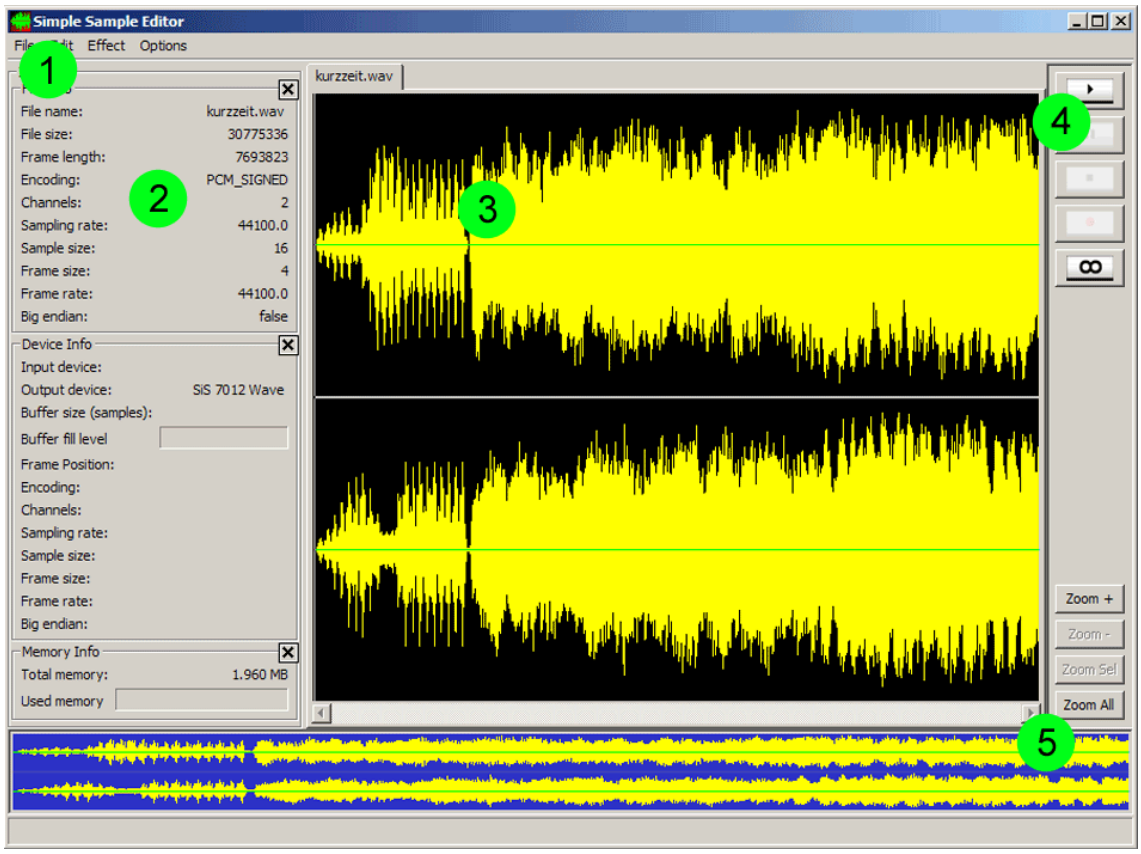

Abbildung 18: Screenshot Sampleeditor Grundversion

1.2.1.1 Die Menü-Funktionen

File

Das File-Menü bietet die gängigen Optionen New, Open, Save, Save As... und Close. Es können beliebig viele Dateien gleichzeitig geöffnet werden, allerdings nicht mehrmals dieselbe Datei. Im Open-Dialog kann jede beliebige Datei (auch mehrere auf einmal) ausgewählt werden.[158] Ist das Format der Datei einem Java Sound PlugIn bekannt, wird diese geöffnet, andernfalls wird eine Fehlermeldung ausgegeben. Dabei kann der Sampleeditor sehr gut vom PlugIn-Konzept von Java Sound profitieren, da für die Unterstützung weiterer Dateiformate lediglich die entsprechenden Java Sound PlugIns in den Classpath eingebunden werden müssen.

Beim erstmaligen Öffnen einer Datei wird zunächst eine Spitzenwertdatei erzeugt, die für eine performantere grafische Darstellung nötig ist. Sie kann wieder verwendet werden, wenn dieselbe Datei später erneut geöffnet wird.

Das Speichern von Dateien geschieht derzeit ausschließlich im WAVE-Format.

Edit

Im Edit-Menü finden sich grundlegende Bearbeitungsfunktionen (siehe Abbildung 19). Sie sind alle nichtdestruktiv realisiert, weshalb jederzeit beliebig viele Schritte wieder rückgängig gemacht wer-

[158] Alternativ dazu können Dateien auch per Drag ´n´ Drop durch Ziehen auf das Anwendungsfenster geöffnet werden.

den können. Dies wird durch Schnittlisten (siehe III.1.3.2.3) realisiert, die intern pro geöffneter Datei verwaltet werden. Jeder Bearbeitungsschritt erzeugt eine neue Schnittliste, die auf der vorherigen aufbaut. Das Rückgängigmachen erfolgt dann durch Entfernen der zuletzt erzeugten Schnittliste. Die abzuspielenden oder darzustellenden Audiodaten werden dann aus den Schnittlisten erzeugt, sobald sie benötigt werden.

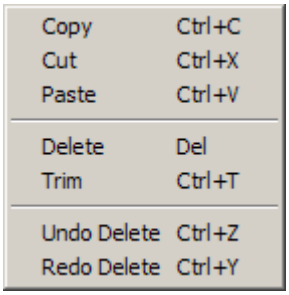

Abbildung 19: Das Edit-Menü

An Edit-Funktionen gibt es zunächst die Elementarfunktionen Copy, Cut und Paste, die das Kopieren, Ausschneiden und Einfügen von Audiomaterial innerhalb des Sampleeditors ermöglichen. Ein Kopieren von und zu anderen Audioanwendungen würde eine plattformabhängige Implementierung erfordern und ist daher derzeit nicht vorgesehen. Bei Copy bzw. Cut wird der selektierte Bereich kopiert bzw. ausgeschnitten. Ist kein Bereich selektiert, sind diese Funktionen auch nicht verfügbar. Paste überschreibt den selektierten Bereich mit dem Inhalt des internen Clipboards, falls dieses Daten enthält. Ist kein Bereich selektiert, wird der Inhalt an der aktuellen Position eingefügt, ohne etwas zu überschreiben.

Die Funktionen Delete und Trim beziehen sich ebenfalls auf den selektierten Bereich und stehen daher nur zur Verfügung, wenn ein Bereich selektiert ist. Delete entfernt den selektierten Bereich, und Trim entfernt alles außerhalb des selektierten Bereichs. Schließlich existieren noch die Funktionen Undo zum Rückgängigmachen des letzten Bearbeitungsschritts sowie Redo zum Wiederherstellen eines vorher zurückgenommenen Bearbeitungsschritts.

Effect

Für das Einbinden von Effekten wurde eine eigene Effektschnittstelle entwickelt, über die relativ einfach neue Effekte hinzugefügt und in den Sampleeditor eingefügt werden können. Ähnlich wie bei Java Sound PlugIns genügt es, eine Subklasse von `com.groovemanager.sampled.fx.Effect` zu implementieren, in den Classpath einzubinden und diese in der Datei `META-Inf/Services/com.groovemanager.sampled.fx.Effect` bekannt zu machen. Beim Erstellen des Menüs wird nach allen Effekt-Implementierungen gesucht, und jede wird in das Effect-Menü als Option mit eingebunden.

Ein Effekt wird stets auf die aktuelle Auswahl angewandt. Ist kein Bereich ausgewählt, wird er auf die gesamte Datei angewandt. Beim Aufrufen eines Effekts erscheint zunächst ein Dialog mit allen vom gewählten Effekt gebotenen Einstellungsmöglichkeiten. Sind hier die gewünschten Einstellungen vorgenommen, kann der Effekt angewandt werden. Auch das Anwenden der Effekte erfolgt nichtdestruktiv. Hierfür wird das Ergebnis der Verarbeitung in eine temporäre Datei gespeichert. Durch das Hinzufügen einer neuen Schnittliste wird dann im gewählten Bereich die ursprüngliche Quelle durch diese Datei ersetzt. Effekte können daher ebenso durch Entfernen der letzten

Schnittliste schnell und einfach rückgängig gemacht werden. Die derzeit implementierten Effekte (Normalize, Pseudo Echo und Chorus/Flanger) haben eher experimentellen Charakter.

Options

Das Options-Menü beinhaltet derzeit lediglich den Eintrag Settings zum Aufrufen einer Konfigurationsseite. Im Settings-Dialog (siehe Abbildung 20) kann der Wiedergabe- und Aufnahmetreiber gewählt werden. Darüber hinaus besteht die Möglichkeit, mit Änderungen der Puffergröße und der Priorität des Player-Threads zu experimentieren. Die Puffergröße bestimmt, wie viele Samples im Voraus gelesen werden. Je größer der Puffer, desto kleiner die Wahrscheinlichkeit eines buffer underruns, also von Aussetzern im Audiodatenstrom. Allerdings steigt auch die Reaktionszeit der Anwendung mit der Puffergröße. Für echtzeitkritische Anwendungen (wie z.B. in der MC-909-Version beim Vorhören der Rhythm Sets, siehe 1.2.2.4) sollte die Puffergröße daher möglichst klein gewählt werden. Dagegen kann beim Aufnehmen ein großer Puffer helfen, Timingschwankungen auszugleichen. Im Info-Bereich wird die tatsächlich vom Treiber verwendete Puffergröße beim Abspielen oder Aufnehmen angezeigt. Bei der Änderung der Thread-Priorität sollte man vorsichtig sein: Eine höhere Priorität bedeutet nicht immer auch besseres Echtzeitverhalten.

Das Ergebnis der in diesem Dialog getätigten Einstellungen kann nicht nur gehört, sondern auch im Info-Bereich (s. u.) – insbesondere anhand des Pufferfüllstatus – beobachtet werden.

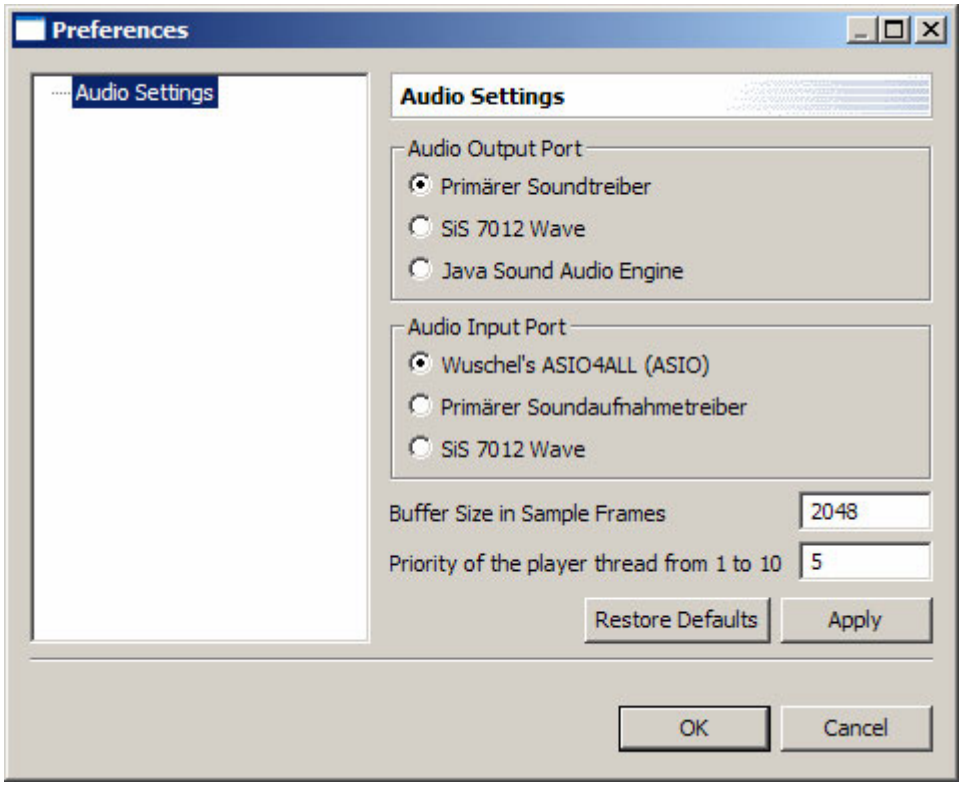

Abbildung 20: Der Settings-Dialog

1.2.1.2 Der Info-Bereich

Der Info-Bereich im linken Teil des Fensters besteht aus drei Gruppen, die jeweils Informationen über einen bestimmten Bereich liefern. Diese können jeweils mit dem ☒-Button rechts oben versteckt und dann mit dem Show ... Info-Button wieder sichtbar gemacht werden.

Folgende Gruppen existieren:

- ***File Info***
  Informationen über Name, Größe und Format der gerade aktiven Datei
- ***Device Info***
  Informationen über Name, Puffer, Position und Format des gerade für das Abspielen oder Aufnehmen verwendeten Treibers
- ***Memory Info***
  Informationen über den aktuellen Speicherverbrauch der JVM-Instanz

1.2.1.3 Wellenformdarstellung

In diesem Bereich kann mit Hilfe der Register zwischen den momentan geöffneten Dateien gewechselt werden. Die Wellenformdarstellung selbst besitzt stets eine Position, eine Auswahl sowie eine beliebige Anzahl von Markern.

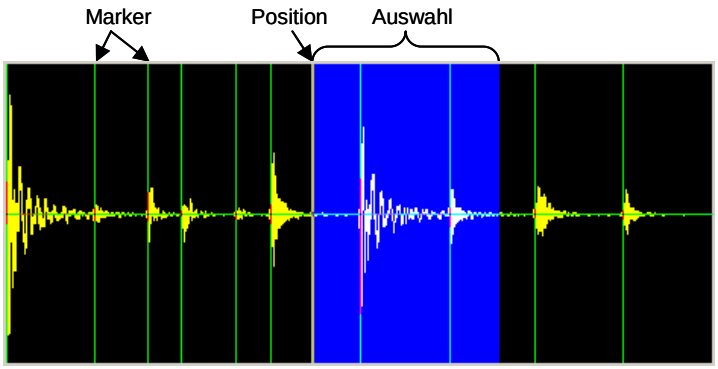

Abbildung 21: Wellenformdarstellung mit Markern, Position und Auswahl

Die Position der Wellenform wird durch einen senkrechten weißen Strich dargestellt und kann durch einfaches Klicken mit der Maus an die gewünschte Position gesetzt werden. Das Abspielen startet immer von der aktuellen Position. Für die Dauer des Abspielens oder Aufnehmens kann die Position nicht verändert werden.

Die Auswahl kann durch Drücken und Ziehen der Maus festgelegt werden. Es ist auch möglich, durch Klicken ohne Mausbewegung die aktuelle Auswahl zu löschen. Abgespielt wird immer der beim Drücken auf den Play-Button selektierte Bereich oder – falls kein Bereich selektiert ist – die gesamte Datei.

Innerhalb der Wellenform können bestimmte Punkte mit Markern markiert werden.[159] Die Marker können entweder aus einer Datei gelesen werden, deren Format solche Markierungen erlaubt oder per Hand hinzugefügt werden. Die Unterstützung für das Lesen von Markern aus Audiodateiformaten war mit ein Grund für die Entscheidung, eine eigene PlugIn-Schnittstelle zu entwickeln. Java Sound hätte nicht garantieren können, dass ein `AudioFileReader`, der diese Marker lesen kann, einem anderen vorgezogen wird, wenn beide das Audioformat der Datei lesen können. Die Marker

[159] Verwendungsmöglichkeiten für Markierungen innerhalb von Audiodateien gibt es zahlreiche. Ein Beispiel dafür findet sich beim Exportieren von Rhythm Sets in der MC-909-Version des Sampleeditors (siehe 1.2.2.4).

werden wie in Tabelle 3 bei den Slices beschrieben als property im `AudioFileFormat`-Objekt gespeichert.

Ein Marker kann hinzugefügt oder entfernt werden, indem bei gedrückter STRG-Taste mit der Maus an die gewünschte Stelle in der Wellenform geklickt wird. Mit gehaltener SHIFT-Taste kann ein Marker selektiert und dann mit der Maus oder mit den Pfeiltasten nach links oder rechts verschoben werden. Hält man beim Drücken der Pfeiltasten die STRG-Taste gedrückt, wird der Marker jeweils nur um ein Sample verschoben.

1.2.1.4 Transportleiste

Rechts oben befindet sich die Transportleiste zur Steuerung des Abspiel- und Aufnahmevorgangs. Bei Betätigung des Play-Buttons wird der gerade selektierte Bereich abgespielt oder – falls kein Bereich selektiert ist – der Bereich von der aktuellen Position bis zum Ende der Datei. Das Abspielen erfolgt stets per Streaming, obwohl Java Sound mit dem `Clip`-Interface auch einen anderen Mechanismus erlauben würde. So wird gewährleistet, dass der Speicherverbrauch niedrig bleibt und dass keine Begrenzung bezüglich der abspielbaren Datenmenge existiert (➔ Skalierbarkeit).

Der Pause-Button ist nur aktiv, wenn gerade abgespielt oder aufgenommen wird. Durch ihn kann der aktuelle Abspiel- oder Aufnahmevorgang angehalten werden; die Treiberressourcen werden dabei aber nicht freigegeben, so dass ein sofortiges Fortfahren möglich ist. Im Gegensatz dazu wird beim Betätigen des Stop-Buttons die Position an den Anfang des selektierten Bereichs zurückgesetzt und alle Ressourcen werden freigegeben, so dass andere Anwendungen wieder darauf zugreifen können.

Die Aufnahme wird mit dem Rec-Button gestartet. Eine Aufnahme ist nur möglich, wenn eine neue Datei mit dem Menübefehl File>New oder dem Tastaturkürzel Strg+N erzeugt wurde. Nachdem eine Aufnahme durch Drücken des Stop-Buttons beendet wurde, kann in diese Datei nicht mehr weiter aufgenommen werden. Stattdessen muss dafür eine neue Datei angelegt werden. Die Inhalte können dann mit Copy/Paste zusammengefügt werden. Beim Betätigen des Rec-Buttons wird in dem Format aufgenommen, das beim Anlegen der neuen Datei angegeben wurde. Ist dies nicht möglich, wird eine entsprechende Konvertierung in den Aufnahmeprozess integriert. Ist auch das nicht möglich, wird der Aufnahmevorgang mit einer Fehlermeldung abgebrochen.

Es sollte beachtet werden, dass die aufgenommenen Daten zunächst im temporären Ordner des Betriebssystems gespeichert und erst beim Beenden gelöscht werden. Eine Auswahl des Temporärverzeichnisses wäre wünschenswert, ist aber derzeit noch nicht implementiert.

1.2.1.5 Scroll- und Zoom-Bereich

Um den sichtbaren Bereich anzupassen, existieren verschieden Möglichkeiten: Zunächst kann mit den Zoom-Buttons rechts unten die Zoom-Stufe bestimmt werden. Dabei steht Zoom Sel für das Zoomen zur aktuellen Auswahl und Zoom All für das Anzeigen der gesamten Datei. Befindet man sich in einer Zoom-Stufe, in der nicht der gesamte Dateiinhalt sichtbar ist, kann mit der Scrollbar unterhalb der Wellenform an die gewünschte Stelle gescrollt werden. Die Buttons Zoom + und Zoom - ermöglichen das Hinein- bzw. Herauszoomen, was alternativ auch mit den Tasten + bzw. - erreicht werden kann.

Eine weitere Möglichkeit, den sichtbaren Bereich zu definieren, bietet die Wellenform am unteren Rand des Editors. Sie stellt immer eine Übersicht über den gesamten Dateiinhalt dar. Durch Selektieren eines Bereichs in dieser Wellenform wird der betreffende Bereich in der Hauptwellenformdarstellung angezeigt. Ebenso dient die untere Wellenform zur optischen Kontrolle, welcher Teil der Datei gerade in der Hauptwellenformdarstellung zu sehen ist.

1.2.2 Die MC-909-Version

Aufbauend auf der Grundversion des Sampleeditors wurde eine weitere Version entwickelt, die die Kommunikation mit der Sampling-Groovebox Roland MC-909 ermöglicht. Sie unterscheidet sich von der Grundversion optisch lediglich in der Nutzung des linken Fensterteils, der in der Grundversion als Info-Bereich dient, sowie in einer neuen Seite für die MIDI-Einstellungen im Settings-Dialog. Außerdem kommt am unteren Rand eine Nachbildung der Keyboard-Pads der MC-909 hinzu (siehe Abbildung 22), die eine Hilfestellung für das Exportieren von Rhythm Sets bieten soll.

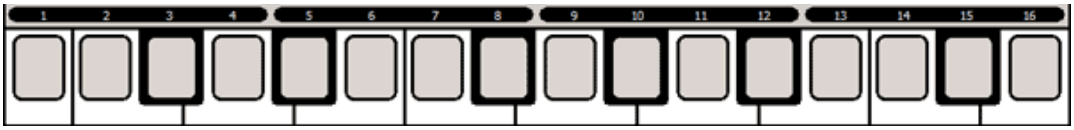

Abbildung 22: Die Keyboard-Pads der MC-909 als Bedienelemente im Sampleeditor

Im Folgenden wird diese Groovebox kurz vorgestellt. Anschließend werden die neuen Elemente der MC-909-Version gegenüber der Grundversion des Sampleeditors beschrieben.

1.2.2.1 Kurzportrait der Groovebox MC-909

Abbildung 23: Roland MC-909

Ende 2003 brachte der Musikinstrumentenhersteller Roland mit dem Modell MC-909 eine Groovebox auf den Markt, die Sampler, Sequencer und Synthesizer in einem Gerät vereint.

Der Synthesizer/Sampler ermöglicht die Klangerzeugung entweder auf Basis der mitgelieferten ROM[160]-Samples oder auf Basis eigener Klangsamples, die über den Audioeingang aufgenommen oder über die USB-Schnittstelle importiert werden können. Die Klänge können entweder als Einzelklang (sog. *Patch*) oder als Zusammenfassung mehrerer Klänge in einem sog. *Rhythm Set* gespeichert werden. Ein Patch kann über die gesamte Klaviatur in unterschiedlicher Tonhöhe gespielt werden. In einem Rhythm Set ist dagegen jeder Taste ein anderer Klang zugeordnet.

Der Sequencer bietet die Möglichkeit, die Klänge über MIDI-Sequenzen auf 16 Spuren anzusteuern. Die Zusammenfassung von je einer gleich langen Sequenz pro Spur wird als *Pattern* bezeichnet.

Eine Besonderheit der MC-909 ist die *Auto-Sync*-Funktion, die den tempounabhängigen Umgang mit Samples ermöglicht. Das bedeutet, dass Samples schneller oder langsamer abgespielt werden können, ohne dass sich dabei die Tonhöhe ändert.

Zur Datenhaltung verwendet die MC-909 einen internen 16 MB Flash-ROM-Speicher sowie optional eine bis zu 128 MB große Smart-Media-Karte. Das Gerät kann in einen USB-Modus versetzt werden, um von einem angeschlossenen Computer auf eines dieser beiden Speichermedien zuzugreifen.

Als Klaviaturersatz bietet die MC-909 16 anschlagdynamische Pads; alternativ oder zusätzlich kann aber auch ein externes MIDI-Keyboard zum Anspielen der Klänge angeschlossen werden.

1.2.2.2 Die Sample-Eigenschaften

Für jedes geöffnete Sample können links unten im Editor MC-909-spezifische Eigenschaften festgelegt werden (siehe Abbildung 24), die später beim Exportieren berücksichtigt werden. Neben dem Namen des Samples ist das zunächst der Loop Mode. Dieser bestimmt, ob das Sample vorwärts, rückwärts, im Loop oder nur einmal abgespielt wird. Der timestretch type (hier mit TS Type abgekürzt) bestimmt das Verhalten dieses Samples im oben erwähnten Auto-Sync-Modus. Ein niedriger Wert führt zu besseren Ergebnissen bei tonalem Material, ein hoher Wert empfiehlt sich dagegen für perkussives Material.

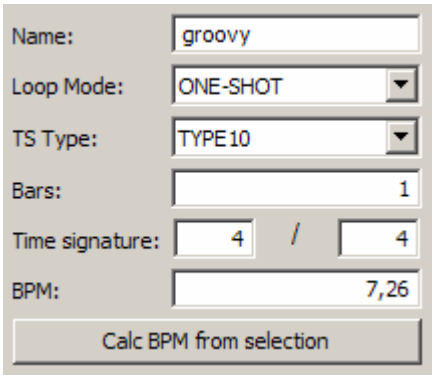

Abbildung 24: Die Sample-Eigenschaften

Die darunter folgenden Elemente dienen alle der Tempoermittlung und können für alle Samples ignoriert werden, die kein bestimmtes Tempo haben. Beispielsweise hat ein Drumloop immer ein Tempo, ein einzelner Drumsound dagegen nicht. Mit Bars und Time signature wird die Länge des

[160] Read Only Memory

Samples in Takten sowie die Taktart angegeben. Das Tempo in BPM (Beats per Minute) kann entweder per Hand eingegeben oder mit dem darunter liegenden Button aus den Angaben von Bars und Time signature sowie der Länge der aktuellen Auswahl in der Wellenformdarstellung errechnet werden.

1.2.2.3 Samples Im- und Exportieren

Unterhalb der Sample-Eigenschaften befindet sich der Import/Export-Button für den Austausch von Samples zwischen Editor und MC-909. Beim Drücken dieses Buttons öffnet sich der Import/Export-Dialog (siehe Abbildung 25).

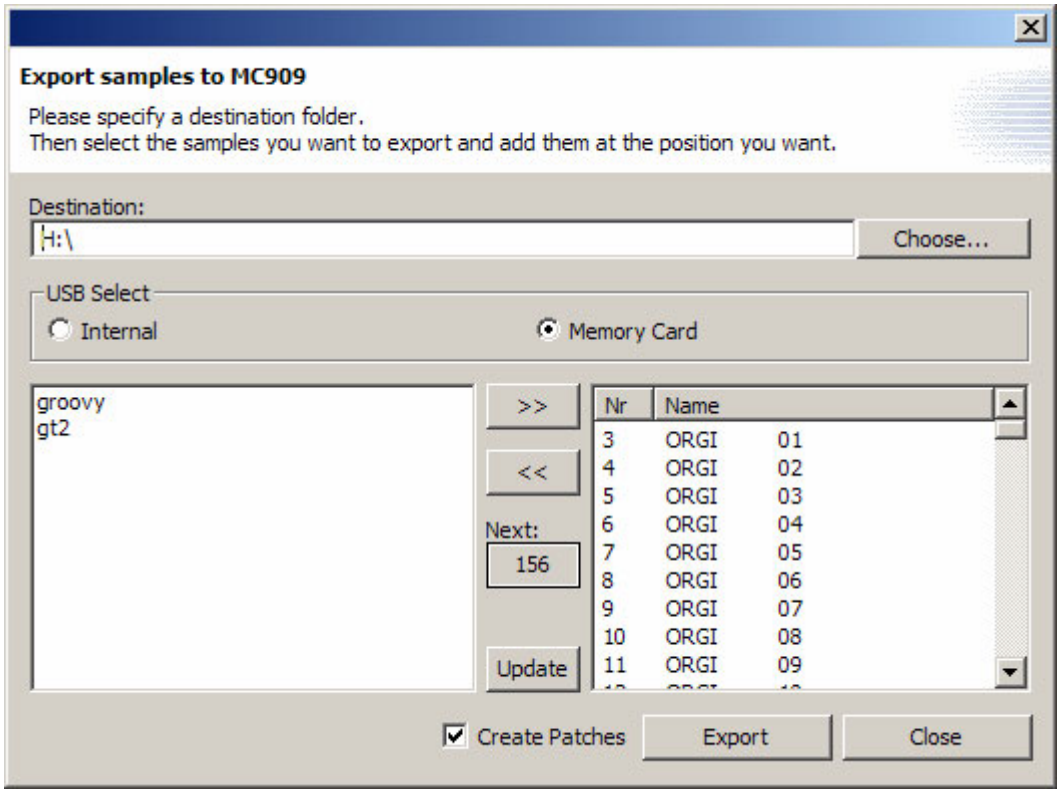

Abbildung 25: Der Import/Export-Dialog

Um Samples mit der MC-909 austauschen zu können, muss diese am Gerät in den USB-Modus versetzt werden. Dabei wird die MC-909 automatisch als eigenes Laufwerk zum Betriebssystem gemountet. Den Pfad zu diesem Laufwerk muss man in dem Dialog unter Destination angeben, damit der Sampleeditor auf das Gerät zugreifen kann[161].

Beim Versetzen der MC-909 in den USB-Modus muss angegeben werden, ob auf den internen Flash-ROM-Speicher oder auf die Smart-Media-Karte zugegriffen werden soll. Diese Auswahl sollte im Dialog unter USB Select wiederholt werden, da sonst das automatische Erstellen eines Patches nicht korrekt funktioniert.

[161] Um den Editor ohne MC-909 zu testen, genügt es, ein Verzeichnis anzulegen, das die Unterverzeichnisse `/ROLAND/SMPL` enthält. Dieses Verzeichnis kann dann unter Destination angegeben werden.

In der linken Auswahlliste findet man alle derzeit im Sampleeditor geöffneten Dateien, auf der rechten Seite eine Liste aller derzeit auf dem gewählten Speichermedium der MC-909 enthaltenen Samples. Jedes Sample ist hier einer Nummer zugeordnet. Stereosamples belegen zwei Nummern.

Mit dem >>-Button können geöffnete Samples zum Exportieren markiert werden. Sie erscheinen dann in der rechten Liste an der Nummer, die unter Next ausgewählt wurde. Existiert an dieser Stelle bereits ein Sample, wird der Benutzer gefragt, ob er dieses überschreiben will. Zum Export markierte, aber noch nicht exportierte Samples erscheinen in der rechten Liste in roter Farbe.

Ebenso können mit dem <<-Button Samples von der MC-909 importiert und im Sampleeditor geöffnet werden. Dabei wird eine lokale Kopie des Samples angelegt, um zu gewährleisten, dass es auch nach Beendigung der USB-Verbindung noch gelesen werden kann. Daher muss das Sample wieder zurückexportiert werden, um vorgenommene Änderungen permanent zu machen.

Der Update-Button dient schließlich dazu, die Liste der Samples auf der MC-909 neu einzulesen.

Um den Export-Vorgang für alle zum Export vorgesehenen Samples auszuführen, muss der Export-Button gedrückt werden. Dabei sollte die nebenstehende Checkbox ausgewählt sein, falls aus den übertragenen Samples automatisch auch Patches generiert werden sollen, was meistens der Fall sein wird. Wurde diese Option selektiert, erscheint nach erfolgtem Export für jedes Sample ein Dialog, in dem der Benutzer aufgefordert wird, den Speicherort für den Patch anzugeben.

Zum erfolgreichen Erstellen eines Patches muss zunächst eine MIDI-Verbindung zur MC-909 hergestellt und der richtige MIDI-Treiber im Settings-Dialog ausgewählt sein. Nach Erzeugen des Patches muss der Benutzer ihn direkt an der Groovebox abspeichern, um ihn zu behalten. Ein ferngesteuertes Abspeichern ist leider nicht möglich. Damit ist der Exportvorgang abgeschlossen.

1.2.2.4 Rhythm Sets Exportieren

Ein Rhythm Set besteht aus 16 unterschiedlichen Klängen, die jeweils einer Taste auf der Klaviatur zugeordnet sind. Diese Tasten entsprechen genau den 16 Keyboard-Pads der MC-909. Mit dem MC-909-Sampleeditor ist es möglich, aus einem Sample, das mit Markern in mehrere Samples unterteilt wurde, solch ein Set zu generieren. Folglich ist der Export Rhythm Set-Button nur dann aktiv, wenn die Wellenformdarstellung mindestens einen Marker enthält. Neben dem Rhythm Set wird auf Wunsch auch noch eine MIDI-Sequenz exportiert, die die Abfolge der Marker widerspiegelt. Dies ist eine Funktion, die oft gewünscht wird, da sie beispielsweise das Arbeiten mit Drumloops erheblich erleichtert.

Das Ergebnis kann vorgehört werden, indem mit der Maus die Keyboard-Pads im Sampleeditor angeklickt werden. Für jedes Pad wird das zugehörige Einzelsample abgespielt. Alternativ zur Bedienung mit der Maus kann auch eine externe MIDI-Klaviatur an den im Settings-Dialog eingestellten MIDI-Eingang angeschlossen werden, z.B. auch die MC-909 selbst. Um das Vorhören möglichst realistisch zu gestalten, ist es wünschenswert, dass die Verzögerung zwischen dem Betätigen einer Taste und dem Abspielen des Samples minimal ist. Dies kann durch die Wahl des richtigen Treibers sowie durch eine möglichst kleine Puffergröße erreicht werden. Es handelt sich hier um eine harte Echtzeitanforderung.

Beim Drücken des Export Rhythm Set-Buttons öffnet sich ein Wizard, dessen Schritte im Folgenden beschrieben werden:

Schritt 1: Zielauswahl

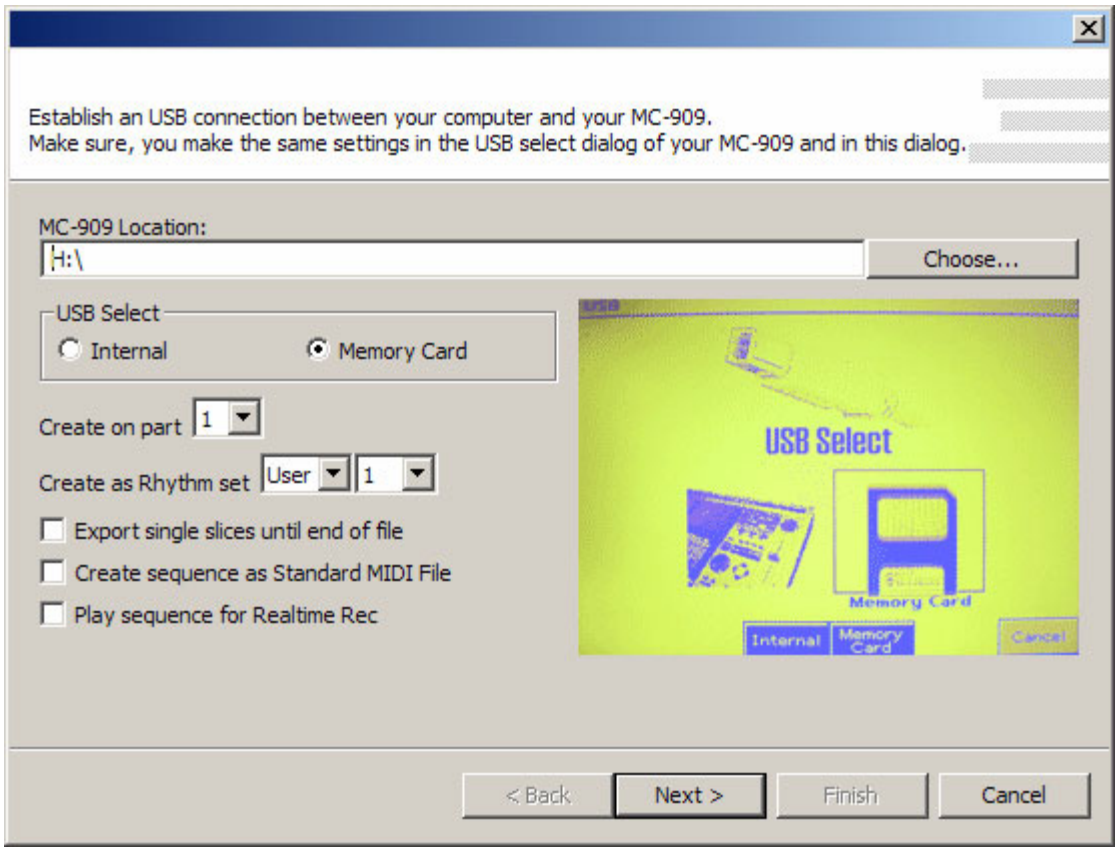

Abbildung 26: Zielauswahl im "Export Rhythm Set"-Wizard

Auf der ersten Seite wird der Benutzer aufgefordert, zunächst wie beim Exportieren eines Patches eine USB-Verbindung mit der MC-909 herzustellen und den Speicherort auszuwählen, an dem das Rhythm Set erstellt werden soll. Darunter stehen die drei folgenden Optionen zur Auswahl:

- ***Export single slices until end of file***
  Diese Option gibt an, dass jedes Einzelsample bis zum Dateiende gehen soll. Ist sie nicht selektiert, endet das Sample dort, wo das nächste Sample anfängt.
- ***Create sequence as Standard MIDI File***
  Mit dieser Option kann die MIDI-Sequenz, die sich aus der Abfolge der Einzelsamples ergibt, in einer MIDI-Datei gespeichert und auf die MC-909 übertragen werden. Diese kann dann später am Gerät in ein Pattern importiert werden.
- ***Play sequence for Realtime Rec***
  Wird diese Option ausgewählt, so wird die MIDI-Sequenz der Einzelsamples über den MIDI-Anschluss an die MC-909 übertragen.

Erst wenn eine USB-Verbindung hergestellt und der richtige Pfad unter MC-909 Location angegeben wurde, kann zum nächsten Schritt gesprungen werden.

Schritt 2: Verlassen des USB-Modus

Nun werden alle Einzelsamples an die MC-909 exportiert. Wurde ein Übertragen der MIDI-Sequenz als Datei eingestellt, so wird diese Datei auch exportiert und der Speicherort wird ausgege-

ben. Der Benutzer wird dann aufgefordert, den USB-Modus auf der MC-909 zu beenden. Außerdem muss eine MIDI-Verbindung mit der MC-909 hergestellt werden, damit das Rhythm Set erzeugt werden kann.

Schritt 3: Übertragen der Sequenz

Der Benutzer wird nun aufgefordert, das erzeugte Rhythm Set zu speichern und die exportierten Samples in den RAM zu laden. Wurde ein Übertragen der MIDI-Sequenz über den MIDI-Anschluss gewählt, kann nun dieser Schritt noch ausgeführt werden. Dafür muss die MC-909 zunächst in den Sync Mode „Slave" versetzt werden, damit sie das Tempo sowie Start und Stop von extern bezieht. Außerdem muss der Realtime Rec Standby Modus gestartet werden, damit die Sequenz auch aufgenommen werden kann. Mit Betätigen des Play-Buttons im Wizard kann die Sequenz nun beliebig oft übertragen werden.

2. Probleme und Lösungen

Ein Problem von Java Sound, das sich für den praktischen Teil als besonders gravierend herauskristallisierte, war die unzufriedenstellende Umsetzung der SPI-Schnittstelle für `AudioFileReader` und `AudioFileWriter` (siehe V.3). Als Workaround für diese Schwäche der SPI-API wurde eine eigene PlugIn-Schnittstelle entwickelt (siehe 1.1.4) und außerdem ein RFE[162] in Suns Bug-Datenbank eingetragen[163].

Weiterhin offenbarten sich hin und wieder kleinere Lücken in der Java Sound Implementierung. So fehlt beispielsweise im Java Sound `Sequencer` die Unterstützung für MIDI Clock. Zur Synchronisation beim Überspielen der MIDI-Sequenz beim Exportieren eines Rhythm Sets musste diese Unterstützung daher selbst implementiert werden.

Eine größere Schwierigkeit stellte die Implementierung des ASIO-PlugIns dar, was größtenteils an den unterschiedlichen Ansätzen von Java Sound und ASIO lag. Ob das Knacksen, das im ASIO-PlugIn auftritt, an Java, Java Sound oder einer anderen Ursache liegt, konnte in der begrenzten zur Verfügung stehenden Zeit nicht herausgefunden werden.

Bei der Umsetzung der Mehrkanalunterstützung von ASIO offenbarte das Java Sound API ebenfalls Schwächen. So ist es nicht möglich, unterschiedliche `DataLine`s eines `Mixer`s mit Namen zu versehen, um beispielsweise unterscheiden zu können, welche Kanäle einer Mehrkanalsoundkarte man ansprechen möchte oder ob man den Analog- oder Digitalausgang nutzen möchte. Ebenso existiert kein Mechanismus zur Unterscheidung zwischen einem Monodatenstrom, der auf die zwei Kanäle einer Stereosoundkarte verteilt werden soll, und einem, der tatsächlich nur auf einem Kanal übertragen werden soll.

Ein weiteres Problem, das sich im praktischen Teil ergab, bestand darin, dass das Verhalten der unterschiedlichen Implementierung – insbesondere zwischen 1.4 und 1.5 – stark variiert. Ebenso verhält sich die Java Sound Audio Engine, die ja in Version 1.5 immer noch zur Verfügung steht, in vielen Anwendungsfällen anders, als die anderen `Mixer`-Implementierungen. Als Ergebnis dieser

---

[162] Request for Enhancement

[163] Vgl. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5061439

Unterschiede kann es vorkommen, dass sich der Sampleeditor bei verschiedenen Java-Versionen und Treibern unterschiedlich verhält.

3. Bewertung / Zusammenfassung

Im Lauf der Arbeit am praktischen Teil wurde schnell klar, dass es nicht möglich sein würde, in der zur Verfügung stehenden Zeit die Grenzen von Java Sound vollständig auszuloten. So beschränkte ich mich darauf, als Ziel die Entwicklung eines Sampleeditors mit Unterstützung für die MC-909 anzustreben und dabei Java Sound so weit wie möglich auf die Probe zu stellen.[164]

Tendenziell lässt sich erkennen, dass mit Java und Java Sound brauchbare Audioanwendungen möglich sind, die weder langsam noch besonders ressourcenhungrig erscheinen. Gleichzeitig können aber nicht alle Anforderungen zu vollster Zufriedenheit erfüllt werden.

Im Folgenden wird noch einmal kurz zusammengefasst, was die Ergebnisse aus dem praktischen Teil für die Erfüllung der unter III.3 aufgestellten Anforderungen bedeuten:

Was die Echtzeitfähigkeit von Java Sound betrifft, so konnte mit dem ASIO-PlugIn nicht zu 100% die Echtzeitfähigkeit erreicht werden, wie sie mit anderen Applikationen, die ASIO verwenden, möglich ist. Die genaue Ursache ist jedoch noch unklar. Dagegen war das Ergebnis der Vorhörfunktion im MC-909-Sampleeditor in Bezug auf die Reaktionsgeschwindigkeit zufriedenstellend. Auf eingehende MIDI-Signale kann der `AudioPlayer` so schnell reagieren, dass sich ein Echtzeitgefühl beim Benutzer einstellt.

Es zeigte sich ebenfalls, dass die erwähnten Schwächen in der Handhabung der `AudioFileReader/Writer`-SPIs für professionelle Ansprüche ein großes Problem darstellen, sofern man die Erweiterbarkeit durch SPI-PlugIns voll ausnutzen möchte.

Nichtdestruktives Arbeiten ließ sich im praktischen Teil gut realisieren. Das betrifft Java Sound allerdings nur in geringem Maße, da es sich hierbei eher um eine Highlevel-Funktionalität handelt, die dank Javas Objektorientierung gut umsetzbar ist.

Was die Nutzung vorhandener Ressourcen betrifft, so wurde beim ASIO-PlugIn die schlechte Unterstützung von Mehrkanalhardware durch das Java Sound API deutlich. Es existiert auch bereits ein Eintrag in Suns Bugdatenbank, der dieses Problem anspricht.[165] Abgesehen davon traten im praktischen Teil keine Mängel bzgl. der Unterstützung vorhandener Ressourcen zu Tage.

Bezüglich der Synchronisation musste lediglich – wie im letzten Kapitel erwähnt – für die Synchronisation externer Geräte über MIDI Clock eine eigene Implementierung geschrieben werden.

Alle weiteren relevanten Anforderungen konnten erfüllt werden.

---

[164] Weiterführend hätten beispielsweise Tests auf unterschiedlichen Plattformen, mit unterschiedlicher Hardware, mit unterschiedlichen GC-Algorithmen o.ä. durchgeführt werden können, um detailliertere Ergebnisse zu erhalten. Diese hätten allerdings den zeitlichen Rahmen dieser Arbeit gesprengt.

[165] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=5067483

VII. Abschließende Bewertung

1. Das Arbeiten mit Java Sound

Die ersten Schritte mit Java Sound erfordern durchaus etwas Gewöhnung. So ist die Namensgebung vieler Elemente zumindest fragwürdig und gerade das `sampled`-Package muss in seiner Komplexität erst einmal erfasst werden. In JDK-Versionen bis 1.4.x kommt hinzu, dass viele Bestandteile fehler- bzw. lückenhaft implementiert sind, was die Verwirrung in der Anfangsphase nicht gerade mindert.

Allerdings existieren mit [JSRESOURCES] und [JS-INTEREST] sehr gute Quellen, die schnell und umfassend kompetente Hilfe bieten und damit den Einstieg erleichtern. Jedoch bleibt auch nach längerer Beschäftigung mit dem API hin und wieder – und meinem Eindruck nach öfter als bei anderen APIs – die Notwendigkeit, bestimmte Dinge in der Dokumentation nachzulesen, weil sie sich nicht von selbst erklären. Dies mag aber auch an der komplexen Thematik der Audioprogrammierung liegen.

Auffällig beim Java Sound API ist die konsequente Lowlevel-Orientierung. Es wird nicht versucht, ein Medienverarbeitungsframework à la JMF zu kreieren, sondern man beschränkt sich auf die elementaren Funktionalitäten für den Zugriff auf Audiohardware. Leider wurde dieser Ansatz in den Anfangszeiten durch die Implementierung wieder etwas aufgeweicht, weil alles auf die Java Sound Audio Engine ausgerichtet war und der Zugriff auf externe Hardware erschwert bis unmöglich gemacht wurde. Mit JDK 1.5.0 ist Sun aber auf dem richtigen Weg. *„1.5 ist die erste Version von Java Sound, die grundsätzlich alle Features implementiert."*[166]

2. Stärken / Schwächen

Die größte Stärke von Java Sound ist, dass es als Java-Framework und – seit Version 1.3 – fester Bestandteil der J2SE von vielen Vorteilen profitiert. Für ein Audioverarbeitungsframework sind das insbesondere die Plattformunabhängigkeit und die umfangreiche Klassenbibliothek für die unterschiedlichsten Bereiche, die in der J2SE enthalten ist. Ebenso wird dadurch eine große Verbreitung sichergestellt. Neben der Installation des JRE[167] ist keine weitere Zusatzinstallation wie z.B. für das JMF nötig.

Eine weitere Stärke von Java Sound sind die SPI-Schnittstellen. Zwar lässt sich an der Umsetzung einiges kritisieren, aber prinzipiell ist das Konzept der Erweiterbarkeit gut gelöst. Bestehenden Anwendungen kann dadurch auf einfache Weise eine Unterstützung für neue Standards hinzugefügt werden.

---

166 Aus [INTERVIEW].

167 Java Runtime Environment

Die konsequente Lowlevel-Orientierung von Java Sound kann ebenfalls als Stärke angesehen werden. Letztendlich bleibt dem Programmierer eine größtmögliche Freiheit, wie er seine Anwendung auf der von Java Sound abstrahierten Schicht aufsetzen möchte.

Nachteile von Java Sound sind in erster Linie die späte Reifung zu einer relativ vollständigen Referenzimplementierung. Java Sound existierte ca. fünf Jahre lang in einer Version, die nur sehr eingeschränkt für anspruchsvollere Anwendungen nutzbar war. In dieser Zeit hat es mit Sicherheit einige potentielle Befürworter und Anwender verloren, die es nun erst einmal zurück zu gewinnen gilt. Ebenso kann es noch einige Zeit dauern, bis sich die J2SE 1.5, die derzeit (Mitte Juli 2004) nur als Beta-Version existiert, verbreitet hat.

Ebenso fällt das umständliche API negativ ins Gewicht. Auch nach dessen intensiver Nutzung im praktischen Teil dieser Arbeit kann noch nicht von einem intuitiven Umgang damit gesprochen werden. Zu irreführend ist die Namensgebung, und zu undurchsichtig sind die Zusammenhänge. Allerdings kann dies als „Schönheitsfehler" abgetan werden, solange die Funktionalität dadurch nicht beeinträchtigt wird.

So sehr Java Sound auch von den Vorteilen von Java profitiert, so hat es doch auch unter dessen Schwächen zu leiden. Die Performance ist dabei nicht das Problem, aber eine Umsetzung harter Echtzeitanforderungen, wie sie im Audiobereich angetroffen werden können, ist nicht immer mit dem gewünschten Ergebnis möglich. Ebenso kann nur auf solche Ressourcen zugegriffen werden, für die in irgendeiner Form eine Java-Anbindung besteht.

Ein weiterer Nachteil, den Java Sound von Java erbt, ist der umständliche und langwierige Verbesserungsprozess. API-Änderungen können nur in Major Releases eingeführt werden und müssen außerdem einen komplizierten Prüfungsprozess durchlaufen. Auch bug fixes werden nicht so schnell realisiert, wie das bei einem gut betreuten Open-Source-Projekt der Fall wäre. Im Gegenzug wird allerdings die von Java bekannte Qualität und Planungssicherheit geboten. Nachteilig ist auch die Situation auf dem Mac: Apple besitzt hier die Verantwortung für die Implementierung und scheint derzeit nicht viel Energie in die Entwicklung einer ausgereiften Java-Sound-Implementierung zu stecken.

Was Java Sound derzeit ebenfalls fehlt, ist eine Art zentrale Registry, beispielsweise auf einer Website. An einer solchen Stelle könnten Informationen über existierende PlugIns, property keys sowie mögliche Werte für die verschiedenen identifizierenden Klassen (z.B. `AudioFileFormat.Type`, `AudioFormat.Encoding`, `Control.Type`) gesammelt werden. Würde allerdings solch eine Registry eines Tages eingeführt, was durchaus denkbar ist, wäre sie eine besondere Stärke von Java Sound. [JSRESOURCES] wäre dafür ein geeigneter Ort.

Ein weiterer Schwachpunkt von Java Sound sind einige grundlegende Designschwächen im `midi`-Package (vgl. [INTERVIEW]), die auch in zukünftigen Versionen schwer zu beheben sein werden, da die Abwärtskompatibilität des API gewahrt werden muss. Teilweise können diese jedoch durch Workarounds umgangen werden.

3. Plattformunabhängigkeit

Plattformunabhängigkeit kann in der Audioprogrammierung generell nur in sehr begrenztem Maß erfüllt werden. Wie sich auch im praktischen Teil dieser Arbeit zeigt, sind viele Schnittstellen nur

für bestimmte Plattformen verfügbar. Jede Plattform hat zudem ein komplett anderes zugrunde liegendes Audio- und MIDI-System. Java Sound schafft hier eine gute Abstraktion. Zudem ist man bei Sun inzwischen bemüht, in den Implementierungen die von der jeweiligen Plattform gebotenen Möglichkeiten auszureizen. In den ersten Java-Sound-Versionen war das weniger der Fall: Man versuchte, mit der Java Sound Audio Engine auf allen Plattformen gleiches Verhalten zu erzielen.[168] Man kann daher sagen, dass Java Sound so viel wie möglich, aber inzwischen auch so wenig wie nötig plattformunabhängig gehalten ist.

Ein Problem bei der Plattformunabhängigkeit ergibt sich jedoch bei der Unterstützung der Mac-Plattform. Wie erwähnt, liegt die Zuständigkeit für die Implementierung von Java Sound für Mac-Betriebssysteme bei Apple. Die bisherige Aktivität von Apple macht da nicht viel Hoffnung für die Zukunft. Allerdings existiert für CoreAudio, das Audio/MIDI-Framework des Mac OS X Betriebssystems, eine Java-Anbindung, wenn auch keine Anbindung an Java Sound. Diese Tatsache sowie die Existenz der SPI-Schnittstellen in Java Sound ermöglichen die Entwicklung einer CoreAudio-Anbindung auch von Drittanbietern. Es bleibt zu hoffen, dass Apple oder ein Drittanbieter hier noch eine vollständige CoreAudio-Unterstützung nachreichen wird.

4. Eignung für studiotaugliche Anwendungen

An dieser Stelle sollen noch einmal die Erkenntnisse aus theoretischer und praktischer Arbeit in Bezug auf die unter III.3 erarbeiteten Anforderungen zusammengefasst werden.

Echtzeitfähigkeit ist nicht unbedingt die Stärke von Java. Weiche Echtzeitanforderungen können zwar meist erfüllt werden, da Java-Anwendungen bei guter Programmierung in der Regel keine erheblichen performancetechnischen Nachteile gegenüber nativen Anwendungen haben. Bei harten Echtzeitanforderungen kommt es dagegen auf die maximalen Pausenzeiten an, welche zwar durch die Wahl des richtigen GC-Algorithmus beeinflusst, aber nicht vollends kontrolliert werden können. Sind harte Echtzeitanforderungen zu erfüllen, muss also abgewogen und getestet werden, ob diese mit der gewählten Java-Umgebung eingehalten werden können. Java Sound selbst ist für Echtzeitfähigkeit gut gerüstet. Einziger Nachteil ist die Veränderbarkeit der `MidiMessage`-Objekte: Beim Versenden müssen diese kopiert werden, was die Aktivität des Garbage Collectors in die Höhe treibt und somit für die Einhaltung von Echtzeitanforderungen zum Problem werden kann.[169]

Für derzeit verfügbare Formate und Standards bietet Java Sound zwar keine besonders umfassende Unterstützung, jedoch ist es mit seinen PlugIn-Schnittstellen darauf vorbereitet. Falls die erwähnten Schwächen der `AudioFileReader/Writer`-Schnittstellen in künftigen Versionen noch ausgemerzt werden und womöglich noch eine zentrale PlugIn-Registry eingerichtet wird, kann dieser Punkt als Vorzug von Java Sound gesehen werden. Bis dahin kann mit den Schnittstellen schon viel erreicht werden, mit den Schwächen muss man zunächst jedoch leben.

Selbiges gilt für die Erweiterbarkeit. Für diese wäre es ebenfalls wünschenswert, wenn das API Unterstützung für die Besonderheiten mancher Treibertypen böte und beispielsweise so etwas wie den unter V.2.1 vorgeschlagenen `MixerListener` vorsähe.

---

168 Vgl. [INTERVIEW].
169 Vgl. [INTERVIEW].

In Punkto Konfigurierbarkeit ist Java Sound gut vorbereitet. Allerdings vermisst man auch hier den Zugriff auf den Konfigurationsdialog des Treibers sowie eine Unterscheidung nach Treibertypen (z.B. `Mixer.Type`).

Nichtdestruktives Arbeiten lässt sich in Java dank seiner Objektorientierung gut realisieren. Java Sound spielt hierfür als Lowlevel-Framework keine Rolle.

Skalierbarkeit ist durch die gute Streaming-Unterstützung gegeben. Lediglich vermisst man `AudioOutputStream`s in Java Sound, die beim Schreiben von großen Audiodateien hilfreich wären.

Das Nutzen vorhandener Ressourcen ist bis auf die von Java gegebenen Grenzen und die schlechte API-Unterstützung für Mehrkanalhardware in Java Sound gut möglich.

Zuletzt noch ein im professionellen Bereich äußerst wichtiger Punkt, nämlich die Synchronisation. Hier offenbaren sowohl API als auch Implementierung noch Verbesserungsbedarf. *„Profifeatures wie Synchronisation sind [in der derzeitigen Implementierung] nicht verfügbar."*[170] Eine umfassende Synchronisation mehrerer Mediendatenströme und Geräte zueinander ist noch nicht möglich. Einige Ansätze bestehen im API, meist sind sie jedoch noch nicht implementiert. Aber selbst, wenn `Mixer.synchronize()` und die MIDI-Synchronisationsmethoden voll implementiert wären, würde man mindestens noch eine Wordclock-Unterstützung sowie eine `DataLine.getLatency()`-Methode vermissen.

Zusammenfassend kann festgestellt werden, dass Java Sound derzeit noch zu viele Schwachstellen aufweist, um den hohen Ansprüchen an studiotaugliche Audiosoftware ganz gerecht zu werden. Der Versionssprung von 1.4 zu 1.5 kann zwar ohne Zweifel als Meilenstein angesehen werden, aber es bleiben noch viele Kritikpunkte. Die derzeitige Aktivität bei der Weiterentwicklung macht jedoch Hoffnung, dass viele dieser Mängel in künftigen Versionen verbessert werden können. Trotz aller Kritik soll noch einmal erwähnt werden, dass Java Sound als plattformunabhängiges Audioframework und als Bestandteil eines großen Plattformstandards in seinem Umfang ohne Konkurrenz ist. Würde Java Sound also konsequent weiterentwickelt, könnte es sich zu einer mächtigen Basis für plattformunabhängige Audioanwendungen – auch für den Studiobereich – entwickeln.

5. Zukunftsausblick

„Das heißt aber nicht, dass die Entwicklung von Java Sound hier aufhört."[171]

Wie gezeigt bietet Java Sound eine gute Basis, auf der sich aufbauen lässt. Der Bedarf für Weiterentwicklungen ist allerdings auch durchaus gegeben. Für den Einsatz als Audioframework für normale Desktopanwendungen ist Java Sound schon jetzt gut gerüstet; für den Studiobereich bedarf es noch einiger Verbesserungen.

Momentan ist bei Sun genau eine Person für Java Sound zuständig, und das nicht einmal zu hundert Prozent[172]. Daraus lässt sich ablesen, dass die Weiterentwicklung von Java Sound noch nicht allzu hohe Priorität bei Sun hat.

---

170 Aus [INTERVIEW].
171 Aus [INTERVIEW].
172 Vgl. [INTERVIEW].

> „Allerdings kann sich hier vieles ändern – zum Beispiel erfahren sowohl JMF als auch Java Sound gerade einen Prioritätsschub durch JDS, das Java Desktop System, und die Allianz mit AMD: Mit anderen Worten, Sun will zurück auf den Desktop, und dazu gehört natürlich Sound.“[173]

Bleibt zu hoffen, dass auch professionelle Anforderungen dabei berücksichtigt werden. Java Sound bietet viel Potential für diesen Bereich, und für meine Person kann ich sagen, dass ich damit gerne weiter arbeiten werde.

In der Java-Sound-Mailingliste ([JS-INTEREST]) haben schon viele Entwickler ihre Zufriedenheit über die endlich erreichte Qualität der Implementierung in der Beta-Version von JDK 1.5.0 zum Ausdruck gebracht. Möglicherweise steigt dadurch auch die Bereitschaft, Java Sound einzusetzen, und damit auch der Druck auf Sun, die Priorität für die Weiterentwicklung von Java Sound zu erhöhen.

Oder anders gesagt: *„was wir hoffentlich nie von Java Sound hören werden ist Stille.“*[174]

[173] Aus [INTERVIEW].
[174] Aus [INTERVIEW].

Literaturverzeichnis

Buchquellen

[BORN] **Born, G.** (2001): *Dateiformate – Die Referenz*, Galileo Press, Bonn.

[BRUNS] **Bruns, K. & Neidhold, B.** (2003): *Audio-, Video- und Grafikprogrammierung*, Fachbuchverlag Leipzig.

[BRÜSE] **Brüse, C.** (1999): *Audio im Computer*, Wizoo Verlag, Köln.

[DAUM] **Daum, B.** (2003): *Java-Entwicklung mit Eclipse 2*, dpunkt.verlag, Heidelberg.

[EIDENBERGER] **Eidenberger, H. M. & Divotkey R.** (2003): *Medienverarbeitung in Java*, dpunkt.verlag, Heidelberg.

[HAIN] **Hain, R.** (2001): *Der Musiker-PC,* Voggenreiter Verlag, Bonn.

[HORSTMANN] **Horstmann, C. S. & Cornell, G.** (2000): *Core Java Band 2 – Expertenwissen,* Markt +Technik Verlag, München.

[LINDLEY] **Lindley, C.A.** (1999): *Digital Audio with Java*, Prentice Hall, New Jersey.

[SCHREIBER] **Schreiber, H.** (2002): *Performant Java programmieren,* Addison-Wesley, München.

[SHIRAZI] **Shirazi, J.** (2000): *Java Performance Tuning*, O´Reilly, Sebastopol.

[WATKINSON] **Watkinson, J.** (2001): *The Art of Digital Audio*, Third Edition, Focal Press, Oxford.

Zeitschriften

[STEWART] **Stewart, I.** (2003): *Ein Vierteljahrhundert Mathematik,* Spektrum der Wissenschaft Mai 2003, S. 24-29

Internetquellen

[BURK] **Burk, P., Polansky, L., Repetto, D., Roberts, M., Rockore, D.**: *Music and Computers*
http://music.dartmouth.edu/~book/index.html
Letzter Zugriff: 15.06.2004

[COREAUDIO] **Apple Computer, Inc.** (2001): *Audio and MIDI on Mac OS X*
http://developer.apple.com/audio/pdf/coreaudio.pdf
Letzter Zugriff: 15.06.2004

[COWELL-SHAH] **Cowell-Shah, C. W.** (2004): *Nine Language Performance Round-up: Benchmarking Math & File I/O*
http://www.osnews.com/story.php?news_id=5602
Letzter Zugriff: 16.06.2004

[DICKMAN] **Dickman, L.** (2002): *A Comparison of interpreted Java, WAT, AOT, JIT, and DAC*
http://www.jadcentral.com/campaign/knowledge_kit/esmertec.pdf
Letzter Zugriff: 15.06.2004

[ESSER] **Esser, F.** (2001): *Java 2 – Designmuster und Zertifizierungswissen*
http://www.galileocomputing.de/openbook/java2/index.htm
Letzter Zugriff: 15.06.2004

[JAVASOUND] **Sun Microsystems, Inc.** (2004): *Java Sound API – Programmer's Guide*
http://java.sun.com/j2se/1.5.0/docs/guide/sound/programmer_guide/contents.html
Letzter Zugriff: 15.06.2004

[JSRESOURCES] *Java Sound Resources*
http://www.jsresources.org
Letzter Zugriff: 29.06.2004

[KIRÁLY] **Király, A.** (2001): *Real Time Specification for Java (RTSJ)*,
http://www.ifi.unizh.ch/~riedl/lectures/rtsj.pdf
Letzter Zugriff: 15.06.2004

[LEWIS] **Lewis, J. P.** (2004): *Performance of Java versus C++*
http://www.idiom.com/~zilla/Computer/javaCbenchmark.html
Letzter Zugriff: 15.06.2004

[MELOAN] **Meloan, M.** (1999): *The Science of Java Sound*
http://java.sun.com/developer/technicalArticles/Media/JavaSoundAPI/
Letzter Zugriff: 15.06.2004

[MICROSOFT] **Microsoft** (2002): *DirectKS Sample Application*, Microsoft Hardware and Driver Central
http://www.microsoft.com/whdc/hwdev/tech/audio/DirectKS.mspx
Letzter Zugriff: 15.06.2004

[MIDI] *MIDI Manufacturer's Association*
http://www.midi.org/
Letzter Zugriff: 15.07.2004

[MUSICLINE] *Musicline Genrelexikon Gregorianik*
http://www.musicline.de/de/genre/lexikon/Klassik/Gregorianik
Letzter Zugriff: 13.07.2004

[NET-LEXIKON] *Net-Lexikon*
http://www.net-lexikon.net/
Letzter Zugriff: 15.06.2004

[PHILLIPS] **Phillips, D.** (2003): *Computer Music and the Linux Operating System: A Report from the Front*
http://mitpress.mit.edu/journals/pdf/comj_27_4_27_0.pdf
Letzter Zugriff: 15.06.2004

[PORTMUSIC] *PortMusic APIs*
http://www-2.cs.cmu.edu/~music/portmusic/
Letzter Zugriff: 16.06.2004

[ULLENBOOM] **Ullenboom, C.** (2002): *Java ist auch eine Insel, 3. Auflage*
http://www.galileocomputing.de/openbook/javainsel3/
Letzter Zugriff: 15.06.2004

[VOGT] **Vogt, M.** (2004): *Die Geschichte ATARI´s*
http://www.atari-computermuseum.de/history.htm
Letzter Zugriff: 15.06.2004

Sonstige Quellen

[ASIO] **Steinberg** (1997): Audio Streaming Input Output Specification, Development Kit 2.0

[BÖMERS] **Bömers, F.** (2000): *Wavelets in real time digital audio processing: Analysis and sample implementations*
Diplomarbeit an der Universität Mannheim
http://www.bomers.de/personal/thesis.pdf

[INTERVIEW] Emailinterview mit Florian Bömers (Anhang A)

[JAVADOC] Java 2 SDK Documentation http://java.sun.com/j2se/1.5.0/docs/api/

[JS-INTEREST] Offizielle Java Sound Mailingliste
http://archives.java.sun.com/archives/javasound-interest.html

[MACMILLAN] **MacMillan, K., Droettboom, M., Fujinaga, I.**: *Audio Latency Measurements of Desktop Operating Systems*
Peabody Institute of the Johns Hopkins University

[RELNOTES] *J2SE, Version 1.5.0 Summary of New Features and Enhancements: Java Sound*
http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#js

[ROLAND] **Roland** (2002): *MC-909 MIDI Implementation*

[SCHNEIDER] **Schneider, D. & Ernst, M.** (2000): *Konzepte und Implementierungen moderner virtueller Maschinen*
Diplomarbeit an der Universität Hamburg, Fachbereich Informatik
http://www.virtualmachine.de/2000-ernst-schneider.pdf

Anhang A – Emailinterview mit Florian Bömers

Das folgende Interview führte ich zwischen dem 23.05.2004 und dem 17.07.2004 mit Florian Bömers per Email. Florian Bömers betreibt zusammen mit Matthias Pfisterer das Open-Source-Projekt Tritonus (http://www.tritonus.org), das sich seit 1999 mit Java Sound befasst sowie die Website http://www.jsresources.org, die eine Sammlung zahlreicher Hilfestellungen rund um das Thema Java Sound bietet. Seit Juni 2001 ist er Zuständiger und alleiniger Entwickler für Java Sound bei Sun Microsystems im kalifornischen Santa Clara.

[MR] Wie gestaltete sich grob die historische Entwicklung von Java Sound? (wichtige Meilensteine, Besonderheiten)

[FB] Schon sehr früh, so zwischen 1.0 und 1.1, hat ein Engineer bei Sun angefangen, eine Sound API zu gestalten. Es ergaben sich aber viele Probleme, die vor allem durch die Plattformunabhängigkeit begründet sind, die unter anderem verlangt (damals mehr als heute): exakt gleiche Funktionalität und gleiches Verhalten auf allen Plattformen. Speziell waren wichtig:

- Wie kann man MIDI Dateien unter Solaris abspielen, das keinen OS oder Hardware Support für MIDI hat?
- Wie kann man erreichen, dass z.B. Applets mehrere Sounds auf einmal abspielen, obwohl man nur eine Soundkarte hat?
- Erweiterbarkeit, plug-ins

Es zeichnete sich also mehr und mehr ab, dass die Entwicklung von Java Sound alles andere als trivial war.

Es wurde dann eine integrierte Sound-Engine von Beatnik lizenziert, die Mixen verschiedener Streams, und MIDI Wiedergabe ermöglichte, und das mit einer plattformunabhängigen C Implementierung.

Parallel dazu wurde die Java Sound API entworfen, mit einem Team von internen und auch externen Entwicklern. Mehrere öffentliche Alpha und Beta Versionen, viele Diskussionen führten schließlich zu der Java Sound API, wie wir sie jetzt kennen.

Zuerst war Java Sound übrigens als optional package gedacht, ähnlich wie JMF, das man unabhängig vom JDK installieren muss. Im letzten Moment wurde entschlossen, dass Java Sound in den „core“ aufgenommen wird, also als ein fester Bestandteil der Java Laufzeitumgebung.

Leider hat der Internet-Boom 1998-2000 dazu geführt, dass nach und nach alle Java Sound Entwickler Sun verließen, und es eine Lücke von einem Jahr gab, in der niemand für Java Sound verantwortlich war.

Ich stieß zu Sun einen Monat vor der Deadline für 1.4, so dass ich nur wenig ausrichten konnte. Da man nur in großen Versionen die API ändern kann, ging das dann erstmalig in 1.5. Ich hatte eine Menge vor: Als ich zu Sun kam, hatte ich eine Liste mit ca. 50 API Änderungen. Umgesetzt wurden davon letztendlich ca. 25 für 1.5: ein kleines Expertengremium aus Sun-Internen und externen

Entwicklern diskutierte im Sommer 2003 die genauen Spezifikationen für die neuen API Elemente. Die Entwicklung wurde durch Matthias Pfisterer unterstützt, der für 3 Monate als Praktikant bei Sun tätig war. Sonst lag die gesamte Entwicklung für 1.4 bis 1.5 bei mir allein. 1.5 ist die erste Version von Java Sound, die grundsätzlich alle Features implementiert. Nur Profifeatures wie Synchronisation sind nicht verfügbar.

[MR] Wie kamst Du zu dem Job als Chefentwickler von Java Sound bei Sun?

[FB] Durch meine Arbeit an Tritonus und mein Engagement in der Java Sound Mailingliste lag es nah, dass ich gefragt wurde, ob ich die Stelle übernehmen will, als meine Vorgängerin Sun verließ. Nachdem ich San Francisco und Umgebung, und Sun's Athmosphäre auf der JavaOne Konferenz 2000 kennenlernte, fiel mir die Entscheidung leicht. Es dauerte dann noch genau ein Jahr, bis ich den Job im Juni 2001 tatsächlich begann, vor allem durch den Visumsantrag verzögert.

[MR] Wie offenbarte sich die Situation rund um Java Sound, als Du den Job antratst?

[FB] Im Großen und Ganzen wusste ich natürlich genau, was auf mich zukam: über 200 offene Bugs, eine Implementierung, die seit 1 Jahr nicht mehr gepflegt wurde, keine weiteren Entwickler, die mir helfen können, und jede Menge offener Wünsche von mir und der Community. Ich wusste mehr als ein Dutzend Wege, Java über Java Sound abstürzen zu lassen... Auch war die Portierung von Java Sound auf Linux sehr schlampig gemacht.

[MR] Welche Bedeutung (Priorität) hat und hatte die Weiterentwicklung von Java Sound bei Sun im Vergleich zu anderen Medientechnologien (JMF, MMAPI) und im Vergleich zu komplett anderen Java-Technologien?

[FB] Aus offensichtlichen Gründen bleibe ich hier ein wenig vage. Grundsätzlich hatte Java Sound auf 3 Ebenen zu leiden:

- Die ursprünglichen Entwickler verließen Sun nach Fertigstellung der ersten öffentlichen Version von Java Sound in JDK 1.3
- Da Java Sound in 1.3 für viele Bereiche nicht einsetzbar war, gab es wenige zahlende Java Sound Lizenznehmer, die für Sun die Priorität erhöht hätten.
- Sinkende Rentabilität von Sun, so dass Priorität auf Technologien gesetzt wurde, die direkt Geld erwirtschafteten.

JMF liegt ungefähr im gleichen Bereich wie Java Sound, obwohl es da mehr direkt zuordnende zahlende Kunden gibt. MMAPI dagegen hat viele Lizenznehmer, und wird deshalb auch mit mehr Entwicklern ausgestattet. Ich arbeite übrigens auch zu 50% an Sun's MMAPI Implementierung, und ich war für das API Design von einigen MMAPI Controls zuständig. Natürlich gibt's da Überschneidungen mit Java Sound, und Ziel für die Zukunft ist es, die low-level Implementierungen von Java Sound, MMAPI und JMF zu vereinheitlichen. JMF setzt allerdings auch auf Java Sound auf.

Allerdings kann sich hier vieles ändern – zum Beispiel erfahren sowohl JMF als auch Java Sound gerade einen Prioritätsschub durch JDS, das Java Desktop System, und die Allianz mit AMD: Mit anderen Worten, Sun will zurück auf den Desktop, und dazu gehört natürlich Sound.

[MR] Welches waren die Hauptziele für die Verbesserung von Java Sound im Rahmen von JDK 1.5.0?

[FB] Hauptziele waren:

- Alle Features der API (bis auf ein paar Profifeatures) zu implementieren (Ports, MIDI i/o, Zugriff auf alle installierten Soundcards, hohe Sampling Rates, 24-bit, Receiver und Transmitter für Sequencer)
- Gleiche Features auf allen Plattformen (z.B. waren Ports und MIDI i/o in 1.4.2 noch nicht für Linux implementiert)
- Bessere Performance, vor allem im Bereich latency (durch DirectAudio)
- Vereinfachung (z.B. AudioSystem.getClip())
- Stabilität erhöhen (Abstürze)
- Die API mit wenigen, aber wichtigen Methoden abrunden (Sequencer.loop, MidiSystem.getSequencer(boolean connected), properties, property file

[MR] Inwieweit wurden diese Ziele erreicht?

[FB] Im Großen und Ganzen wurden alle Ziele erreicht. Der Plan, ogg und mp3 zu unterstützen, fiel der Lizenzierungsproblematik zum Opfer. Für Synchronisierung war nicht genug Entwicklungszeit vorhanden.

Das heißt aber nicht, dass die Entwicklung von Java Sound hier aufhört. Im Gegenteil, erst jetzt kann Sun von einer Referenz-Implementierung sprechen. Und erst jetzt ist es sinnvoll, über wirklich große Änderungen nachzudenken. Matthias Pfisterer und ich haben schon Pläne für ein javax.sound.newmidi package – und das nur halb im Scherz.

Auch Profifeatures wie ASIO, DirectX und ASIO plugins, Klangerzeuger, usw. brauchen mehr API support, damit sie effizient von Java aus benutzt werden können. Zum Beispiel verlangt Mehrkanalaudio nach neuen Klassen/Interfaces, um die nötigen Kanalzuordnungen vornehmen zu können.

[MR] Was sind denn die Schwächen des MIDI package, die Dich und Matthias zu der Überlegung bringen, ein komplett neues MIDI package aufzuziehen?

[FB] Im Allgemeinen sind das typische API Design Fehler, die auftreten, wenn man die Implementierung erst nach Festlegung der API fertigstellt. Ein paar Beispiele:

- Timestamps im MIDI package sind pro Device. Sequencer und MIDI IN port haben also unterschiedliche Timestamps. Wenn man nun einen Receiver und Transmitter verbindet, um von einem externen MIDI port in einen Sequencer aufzunehmen, dann müsste man von der API her mit den Timestamps des MIDI devices im Sequencer aufnehmen, obwohl der natürlich eine ganz andere Zeitbasis hat. In der Sun Implementierung ignoriert also der Sequencer die Timestamps für die Aufnahme. Aber damit geht dann die Möglichkeit zur Auswertung der Timestamps des MIDI IN Ports verloren (es sei denn, man umgeht die Aufnahmefunktion vom Sequencer und programmiert sie selbst).

- Für MidiSystem.getTransmitter und MidiSystem.getReceiver gibt es keinen Anwendungsfall, da man damit nicht auswählen kann, was für eine Art Transmitter und Receiver gewünscht sind, und...
- ...von einem Receiver oder Transmitter kann man nicht das dazugehörige MidiDevice erfragen
- MidiSystem.getSequencer lieferte seit 1.3 einen Sequencer, der implizit mit dem damals einzigen Synthesizer verbunden war. Es war unmöglich, diese Verbindung zu trennen. Um das Problem zu lösen, unter Beibehaltung von Kompatibilität zu existierenden Programmen, wurden MidiSystem.getSequencer(boolean) – um mit getSequencer(false) einen Sequencer zu bekommen, der keine Verbindung hat – und MidiDevice.getReceivers()/getTransmitters() – damit man die entsprechenden Verbindungen mit close() schließen kann – eingeführt.
- Der MetaEventListener verschweigt, von welcher Track-Instanz die jeweilige MetaMessage stammt.
- MidiChannel, Instrument, Patch, SoundbankResource, VoiceStatus sind Klassen, die Designfehler haben und die Funktionalität auf andere Weise wesentlich besser repräsentiert werden könnte. Auch fehlen viele moderne Features.
- Manche Klassen enthalten protected Felder, die von der Implementierung gepflegt und entsprechend verwendet werden müssen, da ableitende Klassen auf diese Felder direkt drauf zugreifen könnten. Beispiel ist das "data" array in MidiMessage, das z.B. für eine ShortMessage nur überflüssigen Overhead bedeutet – ShortMessages könnte man intern bequem und effizient in einem "int" Feld speichern.
- MidiMessage ist veränderbar (mutable). Das führt dazu, dass alle MidiMessages, die versandt werden, geklont werden müssen, was zu hohem GC Aufwand führt. Leider ist aber gerade der GC entscheidend für die Realtime-Performance von Java.

[MR] Wie verhält es sich mit Java-Sound-Implementierungen auf dem Mac? Welche Freiheiten hat Apple bzgl. der Implementierung und an welche Vorgaben müssen sie sich halten?

[FB] Als Lizenznehmer von J2SE hat Apple Zugriff auf den kompletten Source Code von Sun's Implementierungen für Solaris, Linux und Windows. Was Apple daraus macht, ist nicht vorgeschrieben. Allerdings muss Apple das JCK (Java Compatibility Kit) bestehen, bevor es eine Portierung von J2SE vertreiben darf. Das JCK ist eine sehr umfangreiche Sammlung von Tests, die alle API's auf Kompatibilität prüfen. Das umfasst Signaturtests der öffentlichen API's, aber auch funktionale Tests, die die javadoc-Spezifikation exakt überprüfen. Es gibt also keine Qualitätsvorgaben, und nur auf das JCK beschränkt funktionelle Vorgaben. Würde Apple zum Beispiel nur Java Sound in Telefonqualität implementieren, so würde dieser Port immer noch J2SE konform sein. Für Java Sound beschränken sich die Lizenznehmer aber im Allgemeinen darauf, den wenigen nativen low-level Quellcode zu portieren, so dass die gesamte Java Implementierung identisch ist. Bei Sun wird natürlich versucht, möglichst viel plattformunabhängigen, also "shared" Code zu programmieren, so dass es Lizenznehmer einfach haben, und auf die Qualität achten können.

[MR] Worin liegen Deiner Meinung nach die hauptsächlichen Stärken und Schwächen des Java Sound API?

[FB] Stärken:

- Plattformunabhängigkeit
- Es ist Java
- Erweiterbarkeit
- Im offiziellen Java Standard enthalten, keine separaten Downloads nötig

Schwächen:

- Un-intuitive API
- byte Arrays nicht gut geeignet für Bearbeitung von Samples, Optimierungen vielfach nicht möglich (z.B. Notwendigkeit des Klonens von MidiMessages, da sie mutable sind)
- Implementierung zu lange zu schlecht (und immer noch nicht perfekt!)
- Probleme der Vereinheitlichung auf Java Ebene von völlig unterschiedlichen Treibermodellen auf Betriebssystem Ebene (Windows MME, DirectSound, Solaris Mixer, Linux ALSA, OSS)
- Viele neue Technologien nicht erhältlich (3D, ASIO, SoundFont, Effekte, etc.)

[MR] Ist es möglich, vorhandene SPIs in neuere Java Sound Versionen zu integrieren? Oder anders gefragt: Warum benötigt man trotz Version 1.5 immer noch den Tritonus PCM2PCMConverter, um beispielsweise 32 Bit-Unterstützung zu bekommen?

[FB] Leider scheitert das an den Lizenzen: Tritonus ist LGPL. Sun benutzt grundsätzlich weder GPL noch LGPL in Java. Selbst bei Public Domain code muss eine interne Prüfung stattfinden, die häufig aufwändiger ist, als die Implementierung selbst.

[MR] Wie schätzt Du die Konkurrenzsituation für Java Sound als plattformübergreifendes Lowlevel-Audio-Framework sowohl innerhalb als auch außerhalb von Java ein? Wie steht es z.B. mit PortAudio?

[FB] Java Sound hat den großen Vorteil, fester Bestandteil eines wichtigen Plattformstandards zu sein. Der größte Nachteil ist die mangelnde Durchsetzung (siehe oben).

[MR] Wie gut ist Java Sound Deiner Meinung nach derzeit (JDK 1.5) für die Entwicklung studiotauglicher Audioanwendungen geeignet? Wo siehst Du Grenzen?

[FB] 1.5 ist die erste Version, die studiotaugliche Anwendungen ermöglicht. Manche Bereiche sind allerdings ungeeignet, vor allem das schon mehrmals erwähnte Thema Synchronisation. Bei MIDI kann man das zumindest selbst programmieren. Auch die Synchronisation von MIDI zu Audio dürfte mittlerweile wesentlich besser gehen als noch mit 1.4.2. Mehrkanalaudio ist nur rudimentär möglich (nämlich alle Kanäle auf einmal). Auch fehlt komplett 3D Audio, was immer wichtiger auch für Studioanwendungen wird. Weiterhin wäre es für einige Bereiche nötig, bessere Einflussnahme auf die Hardware zu haben.

[MR] Wie wahrscheinlich ist es, dass eines Tages das bestehende Java Sound API durch eine neue Version ersetzt wird, in der ein paar konzeptionelle Schwächen ausgeräumt werden und die Namensgebung an ein paar Stellen überdacht wird?

[FB] Grundsätzlich können Klassennamen und Methodennamen nicht geändert werden. Das wird also nicht passieren. Für 1.6 sind aber weitere Vereinfachungen im Stil von AudioSystem.getClip() geplant. Und, wie schon oben erwähnt, mit genug externer Mitarbeit könnte man auch überlegen, ein komplett neues MIDI package zu erstellen. Für den sampled Bereich sehe ich nicht die Notwendigkeit. Dort reicht es, Interfaces und Klassen hinzuzufügen.

[MR] Welche Verbesserungen oder Erweiterungen sind als nächstes für Java Sound geplant? Und was wird es wohl mit Java Sound nie geben?

[FB] Da ich schon vielfältig auf geplante Änderungen eingegangen bin, dazu hier nicht mehr viel. Grundsätzlich kann jeder RFE's (request for enhancement) an Sun abgeben. Auch muss für größere API Änderungen ein Expertengremium gebildet werden. Wenn es um viele Klassen oder ganze Packages geht, muss ein JSR auf jcp.org durchgeführt werden, was prinzipiell auch von Jedermann getan werden kann (allerdings unter erheblichem finanziellen Aufwand, es sei denn man ist eine open source Organisation). Da es wahrscheinlich ist, dass Sun nicht in ein JSR für Java Sound investieren wird, haben Matthias und ich schon überlegt, ob wir das im Rahmen des Tritonus Projekts machen wollen. Das sind aber alles nur Vermutungen, es kann natürlich alles ganz anders kommen, z.B. durch JDS (siehe oben).

Was es nie geben wird? Ich bin gegen high-level Funktionen, wie sie von einigen Anwendern gewünscht werden, dazu gehört im Großen und Ganzen alles, was man in Java selbst programmieren könnte. Besser ist dafür ein Java optional package geeignet, wie JMF. Und was wir hoffentlich nie von Java Sound hören werden ist Stille.

Anhang B – Der Inhalt der beiliegenden CD-Rom

Die beiliegende CD-Rom enthält sämtliche Quelldateien der im praktischen Teil entwickelten Programmteile. Ebenso befindet sich darauf die Dokumentation als Javadoc aller Packages sowie die fertigen PlugIns und Anwendungen. Für Windows und Linux existieren Installer, für andere Plattformen besteht die Möglichkeit, die Installation manuell durchzuführen (siehe Anhang C).

Die auf der CD-Rom befindlichen Dateien sind auch online unter http://www.groovemanager.com/manudiplom/CDRom/ verfügbar.

Die Verzeichnisstruktur ist folgendermaßen aufgebaut:

`/Quellcode`	Sämtliche Quellcodes des praktischen Teils
`/Quellcode/Java`	Java-Quellcode aller Packages
`/Quellcode/Nativ`	C++-Quellcode der JNI-Anbindungen
`/Quellcode/Nativ/jsasio`	C++-Quellcode der JNI-Anbindung für den `ASIOMixerProvider`
`/Quellcode/Nativ/jsrex`	C++-Quellcode der JNI-Anbindung für den `REXFileReader`
`/Javadoc`	Javadoc aller Packages
`/PlugIns`	Kompilierte Versionen der PlugIns
`/PlugIns/jsasio`	Kompilierte Version des `ASIOMixerProvider`
`/PlugIns/jsrex`	Kompilierte Version des `REXFileReader`
`/PlugIns/floatconverson`	Kompilierte Version des `FloatConversionProvider`
`/Sampleeditor`	Kompilierte Versionen der Sampleeditoren
`/Sampleeditor/Grundversion`	Kompilierte Version der Grundversion des Sampleeditors
`/Sampleeditor/MC-909-Version`	Kompilierte Version der MC-909-Version des Sampleeditors
`/Beispielsounds`	Einige Beispielsamples in unterschiedlichen Formaten

Anhang C – Installationsanleitung Sampleeditor

Auf der beiliegenden CD-Rom (siehe Anhang B) befinden sich im Verzeichnis `/Sampleeditor` Installationsdateien für Windows und Linux sowie sämtliche für eine manuelle Installation benötigten Dateien. In diesem Anhang wird der Installationsprozess detailliert beschrieben. Diese Anleitung gilt analog für beide Versionen des Sampleeditors.

Unabhängig von der Installationsart muss zunächst ein Java Runtime Environment (JRE) auf dem System installiert sein. Es wird dringend empfohlen, aufgrund der vielen Verbesserungen von Java Sound J2RE 1.5 zu verwenden, auch wenn es derzeit (Anfang August 2004) nur als Betaversion verfügbar ist. Eine ältere JRE-Version (mindestens 1.3) müsste aber auch funktionieren, allerdings mit einigen Einschränkungen und Bugs. Downloadlinks finden sich unter http://java.sun.com/ bzw. unter http://www.apple.com/java/.

Installation unter Windows

1. Die jeweilige `.zip`-Datei sollte neben der Installationsanleitung noch die Dateien `install_sse_win.jar`, `launcher.ini`, `setup.exe` sowie `setup.manifest` enthalten. Die Installation wird durch den Aufruf von `setup.exe` gestartet.
2. Im nun folgenden Installationsdialog können nun das Installationsverzeichnis sowie die zu installierenden Zusatzpakete ausgewählt werden.
3. Nach erfolgter Installation kann das Programm mit der `.exe`-Datei im gewählten Installationsverzeichnis gestartet werden.

Installation unter Linux

1. Es empfiehlt sich, mit dem Befehl `export PATH=/usr/java/jre/bin:$PATH` die Java-Ausführungsdatei in die `PATH`-Umgebungsvariable mit aufzunehmen, wobei anstelle von `/usr/java/jre/bin` der entsprechende Pfad zur Java-Installation angegeben werden muss. Dieser kann von System zu System variieren. Ansonsten muss bei jedem der folgenden `java`-Aufrufe der gesamte Pfad zur ausführbaren `java`-Datei angegeben werden.
2. Die jeweilige `.zip`-Datei sollte neben der Installationsanleitung noch die Datei `install_sse_linux.jar` enthalten. Mit dem Befehl `java -jar install_sse_linux.jar` kann diese ausgeführt werden.
3. Im nun folgenden Installationsdialog können das Installationsverzeichnis sowie die zu installierenden Zusatzpakete ausgewählt werden.
4. Nach erfolgter Installation kann das Programm mit dem Befehl `java -Djava.library.path=. -jar start.jar` im gewählten Installationsverzeichnis ausgeführt werden.

Manuelle Installation auf beliebiger Plattform

Sollten bei der automatischen Installation Fehler auftreten oder soll der Sampleeditor auf einer anderen Plattform installiert werden, kann die Installation Schritt für Schritt manuell durchgeführt

werden. Theoretisch müsste der Sampleeditor auf allen Plattformen funktionieren, für die eine SWT-Implementierung existiert.

1. Zunächst muss ein Installationsverzeichnis für den Sampleeditor angelegt werden, in welches sämtliche Dateien aus der `.zip`-Datei kopiert werden.

2. Als nächstes müssen die SWT-Bibliotheken für die gewünschte Plattform installiert werden. Diese können von der eclipse-Homepage (http://www.eclipse.org/) bezogen werden. Die im `.zip`-Archiv enthaltenen Dateien gehören in das `/ext`-Verzeichnis des Installationspfads, die nativen Bibliotheken direkt in den Installationspfad selbst.

3. Zusätzlich zu SWT wird noch die JFace-Bibliothek benötigt. Dafür muss die eclipse-Plattform, die ebenfalls von der eclipse-Homepage zu beziehen ist, komplett heruntergeladen werden. Darin befindet sich im Verzeichnis `/plugins/org.eclipse.jface_*.*.*` die Datei `jface.jar`. Welche PlugIns aus der eclipse-Plattform aus Abhängigkeitsgründen mitinstalliert werden müssen, ist versionsabhängig und kann in der Datei `plugin.xml` im Abschnitt `<requires>` nachgesehen werden. Alle `.jar`-Dateien der benötigten PlugIns müssen dann ins `/ext`-Verzeichnis kopiert werden.

4. Das `FloatConversionProvider`-PlugIn ist bereits in den kopierten Dateien enthalten. Die beiden anderen PlugIns (`REXFileReader` und `ASIOMixerProvider`) können bei Bedarf zusätzlich installiert werden, allerdings nur, wenn die Installation auf einem Windows-Betriebssystem stattfindet. Dazu müssen die `.jar`-Dateien in das `/ext`-Verzeichnis und die `.dll`-Dateien direkt ins Installationsverzeichnis kopiert werden.

5. Um weitere Java Sound PlugIns hinzuzufügen, sollten diese sinnvollerweise auch ins `/ext`-Verzeichnis kopiert werden. Jede hinzugefügte `.jar`-Datei muss dann der Classpath-Angabe in der Datei `/META-INF/Manifest.MF` innerhalb der Datei `start.jar` hinzugefügt werden.

6. Nun kann das installierte Programm mit dem Befehl `java -Djava.library.path=. -jar start.jar` im angelegten Installationsverzeichnis ausgeführt werden.
   Hinweis: Für Mac OS X muss die ebenfalls in der eclipse-Plattform enthaltene Datei `java_swt` für die Ausführung benutzt werden, da SWT-Applikationen mit dem normalen `java`-Aufruf unter Mac OS X nicht korrekt funktionieren. Der Aufruf heißt dann entsprechend `java_swt -Djava.library.path=. -jar start.jar`.